IN-62
88784
p 61

NASA Contractor Report 189607

MOVING FORMAL METHODS INTO PRACTICE: VERIFYING THE FTPP SCOREBOARD: PHASE 1 RESULTS

Mandayam Srivas
Mark Bickford

ORA CORPORATION
301 Dates Drive
Ithaca, NY 14850-1313

Contract NAS1-18972
May, 1992

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Abstract

This report documents the Phase 1 results of an effort aimed at formally verifying a key hardware component, called *Scoreboard*, of the Fault Tolerant Parallel Processor (FTPP) being built at Charles Stark Draper Laboratory (CSDL). The Scoreboard is part of the FTPP virtual bus that guarantees reliable communication between processors in the presence of *Byzantine* faults in the system. The Scoreboard implements a piece of control logic that approves and validates a message before it can be transmitted. The goal of phase 1 was to lay the foundation for the Scoreboard verification. We developed formal specifications of the functional requirements and a high-level design of the Scoreboard. We used a preliminary Scoreboard design developed at CSDL as a basis for developing our hardware design. We proved a main correctness theorem for the Scoreboard design from which the functional requirements can be established as corollaries, The goal of Phase 2 is to verify CSDL's final detailed design of the Scoreboard. This task is being conducted as part of a NASA-sponsored effort to explore integration of formal methods in the development cycle of current fault-tolerant computer architectures being built in the aerospace industry.

Keywords: formal requirements specification, fault tolerant parallel computer, Byzantine resilience, computer-aided hardware verification, theorem prover-based verification.

Contents

List of Figures

1 Introduction

Formal verification is a method of validating computer designs and programs by applying symbolic logic techniques. Hardware verification is formal verification applied to the validation of hardware designs. To formally verify a design, one specifies the design and the requirements that the design is expected to satisfy in a formal logical notation. Then, one constructs a formal proof that the design meets the stated requirements. To construct a proof, one needs to use a system for symbolic manipulation, such as a theorem prover, that manipulates expressions belonging to the logic used in the specifications.

The advantage of formal verification over traditional validation methods, such as simulation and testing, is that formal verification gives total test coverage for the verified property. When one constructs a formal proof of a property, the property is shown to hold for all permissible inputs and initial conditions of a design. Of course, there is no guarantee that the requirements specification with respect to which the design is verified is what one wants. But, formal verification brings potential errors in a design into sharper focus by subjecting the design to more intense scrutiny than is possible in simulation. The process of formal verification is, in general, labor intensive. But, there are certain application domains, such as digital hardware, where recent experience [13, 7, 4] suggests that the technology is applicable to industry-scale designs, and certain application areas, such as safety-critical and mission-critical systems, where the advantages of formal verification outweigh its cost.

NASA Langley Research Center has recently initiated a concerted effort [3] involving several organizations, including ORA Corporation, to study the use of formal verification as a possible validation technology for fault-tolerant digital flight-control systems in an application area that requires ultra-high reliability and availability. The first step in this effort was concerned with demonstrating the application of formal verification to key design problems in this area. Some of the significant case studies that were completed as part of the first step are formal verification of a clock synchronization algorithm framework [11], an interactive consistency algorithm [1], and a Byzantine-resilient microprocessor system [15].

One of the goals of the second step of the NASA effort is to explore the integration of formal methods into the design and verification of key components of current fault-tolerant architectures being built in the aerospace industry. Toward this goal, ORA is currently teamed with Charles Stark Draper Laboratory (CSDL), which is one of the organizations in the forefront of building fault-tolerant systems for safety-critical applications. As part of a

NASA/Army-sponsored project, called Army Fault-Tolerant Architecture (AFTA), CSDL is developing the Fault-Tolerant Parallel Processor (FTPP). One of ORA's current tasks is to formally specify and verify a key component, called Scoreboard, of FTPP. The Scoreboard is part of the FTPP virtual bus that guarantees reliable communication between processors in the presence of physical faults in the system. The virtual bus design is based on a conservative fault model, called the *Byzantine* fault model, in which a faulty component can exhibit arbitrary and malicious behavior. The Scoreboard implements a piece of control logic that approves and validates a message before it can be transmitted. The verification of the Scoreboard is being performed in two phases. This report documents the results of the Phase 1 effort.

Since the detailed design of the Scoreboard was still evolving at the time we started our task, our main objective in Phase 1 was to lay the foundation for the verification effort. In Phase 2, we plan to verify the actual CSDL design using the foundation developed in Phase 1. Specifically, the Phase 1 goals were the following:

1. Understand the FTPP architecture and the role of the Scoreboard in FTPP.
2. Formulate a set of abstract requirements on the functional behavior of the Scoreboard. The first two goals were accomplished in consultation with the personnel involved in the development of FTPP.
3. Develop a high-level design for the Scoreboard based on the initial design provided by CSDL and formally specify the design.
4. Formally verify that our Scoreboard design satisfies some of the stated functional requirements.

1.1 Overview of Results

We summarize the Phase 1 accomplishments below.

Requirements Specification: We developed a formal specification of the functional requirements for the Scoreboard in two forms. The *constraints form* describes the functional requirements as a set of constraints on the actual behavior of the Scoreboard. The *functional form* defines the expected behavior of the main operation of the Scoreboard

by means of a function that maps the initial state of the Scoreboard to its final state and the sequence of all outputs produced during the execution of the operation. Since the behavior of the Scoreboard is inherently procedural, the functional form of specification is involved and not as declarative as one might prefer a requirements specification to be. The constraints form is more declarative and easier to read.

Design Specification: We developed a synchronous block level hardware design for the Scoreboard and a formal specification of the design. The Scoreboard design is structured as a finite state machine controller and a set of component blocks that are controlled by the controller. Most of the components are at the level of registers; three of the blocks are substantially more complex than a register and support multiple operations. Our design was based on the algorithmic specification and the VHDL behavioral description of the Scoreboard given in Morton's Master's thesis [9]. We constructed the Scoreboard design and its specification using Spectool, a hardware design verification tool developed at ORA. The design specification defines the behavior of the design for a single clock cycle.

Verification: We mechanically proved a main lemma (`Msglemma`) that establishes a correctness relationship between the design specification and the second form of the requirements specification. The assertions defined in the constraints form of the requirements specification can be proved as corollaries from a stronger version of `Msglemma`. The requirements expressed in the constraints form were not verified in Phase 1. To decompose the proof of `Msglemma` into smaller and more manageable parts, we introduced two additional levels of specification—step level and macro step level— between the design specification and the requirements specification. The step and the macro step levels describe the behavior of the Scoreboard over longer periods of time than the design specification. We proved `Msglemma` with the help of several other lemmas that relate the various levels in the specification hierarchy. We also proved all the helping lemmas used in the proof of `Msglemma`.

1.2 Organization of the Report

This report assumes that the reader has a basic understanding of the issues involved in the design of Byzantine-resilient architectures. It also assumes some familiarity with the steps involved in a formal verification process and a functional style of specification.

The report is organized as follows. The next section briefly describes the tools used in the verification effort. Section 3 gives an overview of FTPP and the role of the Scoreboard in the FTPP design. Section 4 gives the functional requirements specification of the Scoreboard. Section 5 describes the Scoreboard design and its specification generated by Spectool. Section 6 describes the hierarchy of specification and lemmas used in the proof of `Msglemma`. Section 7 summarizes the techniques that were used to prove the major lemmas in the verification. The last section gives our concluding remarks.

2 The Verification Tools Used

In this effort, we used two verification tools, Spectool [12] and Clio [2], both of which were developed at ORA. These tools are summarized in the following subsections. A detailed description of Spectool is given in [14] and that of Clio is given in [2].

2.1 Spectool Summary

Spectool is a computer-aided verification tool targeted for synchronous hardware designs described at a level of representation that is comparable to the register-transfer level of hardware. The tool reduces the effort required for verifying using Clio a design in the targeted class by automating most of the routine, but cumbersome, parts of the verification process. We designed Spectool based on our experience in using Clio to verify several hardware designs, the largest of which was the MiniCayuga processor design.

Spectool provides a window-based graphical user interface that the designer uses to draw a circuit diagram of a hardware design. The designer then annotates the circuit diagram with various pieces of useful information; for example, one piece of information is a set of conditions that are expected to hold at specified points during the operation of the circuit. The tool uses the annotated circuit diagram to automatically generate specification and verification conditions for the circuit. Spectool aids construction of the proof for the conditions by generating a set of control annotations for the prover. The control annotations assist in constructing a proof for a class of verification conditions either automatically or by applying a standard proof strategy. The class of circuit properties targeted by Spectool denote conditions that relate the states of a circuit that are a fixed (not indefinite) distance

apart in time. The tool provides a facility that the designer can use to display the results of the prover in terms of the symbols and concepts introduced by the designer at the circuit diagram level.

2.2 Clio Summary

Clio is a system for proving properties about programs written in an executable functional language, Caliban. Caliban is a higher order, polymorphic, lazy functional language similar to Miranda[1]. A property to be proved is expressed in the Clio assertion language as an arbitrary first-order predicate calculus formula built from atomic literals. An atomic literal is an equation on Caliban expressions. An equation is interpreted to be true if and only if the two expressions in the equation have the same meaning in a domain theoretic semantics defined for Caliban. The first-order formulas have the classical interpretation. In the present application, the specifications of the system and its components are expressed in Caliban. The verification conditions to be proved are expressed in the assertion language.

The basic proof technique of the Clio prover is *normalization*, i.e., simplifying expressions on the two sides of an equation to a common form to prove the equation. The prover supports a set of *proof tactics* that are useful in conjunction with normalization to prove more complex formulae. Some of the proof tactics available are *case analysis*, *structural induction*, *fixpoint induction*, and *proof by contradiction*. The user can use the prover in interactive or automatic mode. In the interactive mode, Clio prompts the user with the available choices in proof tactics, and the user makes the appropriate selection until Clio proves the formula or discovers a contradiction. In the automatic mode, it makes the selection on its own, based on a built-in strategy. The Clio prover is similar to the Boyer-Moore prover in its proving style. The logic of Clio is more expressive than the Boyer-Moore logic because Clio supports unrestricted quantification and higher order functions. It is possible to define and reason about partial functions in Clio because of the lazy semantics of Caliban.

[1]Miranda is a trademark of Research Software Limited.

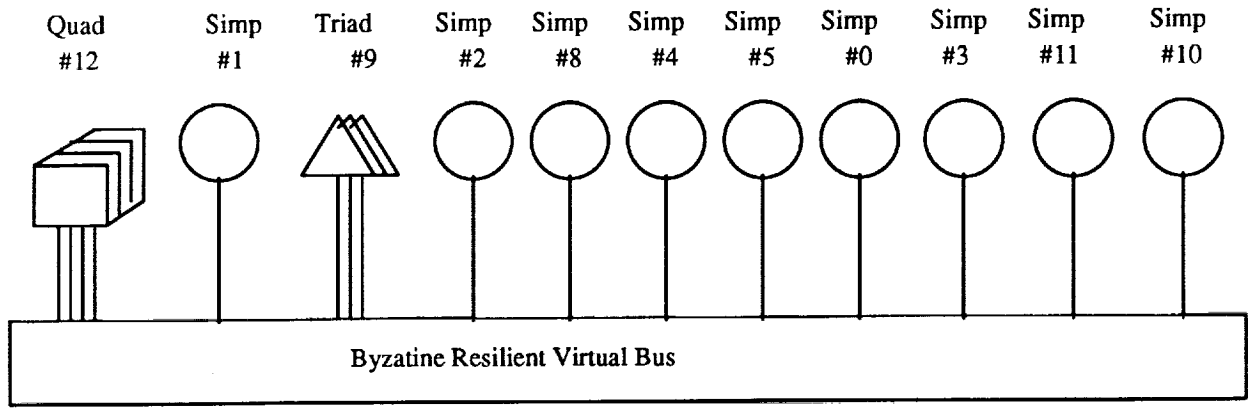


Figure 1: The FTPP Logical Configuration

3 The Scoreboard and Its Environment

The Scoreboard is a key hardware component of the fault-tolerant system being built under the Army Fault Tolerant Architecture (AFTA) project. The AFTA system, designed by CSDL, combines the disciplines of *Byzantine-resilient* fault-tolerant computing and parallel processing. Byzantine resilience design principles have been used to satisfy an ultra-high reliability requirement, while parallel processing capability has been used to meet a high throughput requirement on the system. A Byzantine-resilient system is one that is designed to tolerate arbitrary and malicious faults on the part of the failed components. This type of fault, known as a *Byzantine fault*, may include stopping and then restarting execution at a future time and sending conflicting information to different destinations. In the following subsection, we give an overview of the AFTA hardware architecture and its main components. The following overview is based on the detailed description of AFTA given in [5].

3.1 AFTA Hardware Architecture Overview

The hardware architecture of AFTA is based on the Fault Tolerant Parallel Processor [6], also developed by CSDL. Figure 1 shows a logical view of the FTPP hardware architecture. FTPP is composed of a set of Processing Elements (PE) that are interconnected by means of a specially designed Byzantine-resilient *virtual bus*. The multiple processing elements are used for parallel processing as well as for providing hardware redundancy for fault tolerance. The PEs in the system are partitioned into a set of *virtual groups* (VGs), each of which may contain one (*simplex*), three (*triad*) or four (*quad*) PEs. For example, the configuration shown

in Figure 1 consists of a quad (denoted by a group of squares), a triad (denoted by a group of triangles), and several simplexes (denoted by circles). Virtual groups are logical views of the processing resources capable of accepting work in parallel. When operating redundantly, each processor within a virtual group executes tasks that are functionally congruent with the other members in the group. Fault tolerance for a particular processing resource is ensured by making the virtual group implementing the resource a triad or quad. The architecture is capable of dynamically reconfiguring its ensemble of virtual groups.

The virtual bus is constructed as a fully connected ensemble of specially designed identical hardware components called *network elements* (NE). The NEs implement the inter-PE communications and the redundancy management required by FTPP. A PE, which is a processor with its own memory, just subscribes to a NE. The NEs are replicated because Byzantine resilience requires that the hardware managing redundancy must itself be replicated for fault tolerance. The virtual bus implemented by the group of network elements provides a *Byzantine-Resilient Virtual Circuit* [5] communication abstraction that has the characteristics enumerated below. These characteristics are guaranteed to hold even under Byzantine faults as long as the number of faults in the system is bounded by an amount determined by the redundancy levels of the PEs and the NEs.

1. *Reliable delivery*: Every virtual group that sends a message can expect delivery of the message, assuming the recipient exists.

2. *Order preservation*: Messages sent by one group to another are delivered in the order sent. Specifically, non-faulty members of the recipient group receive messages in the order sent by the non-faulty members of the source group.

3. *Group consensus*:

   - Each nonfaulty member of a group will receive identical copies of the message delivered to the group.
   - All non-faulty members of a recipient group receive messages in identical order.

4. *Synchronous operation*: The absolute times of arrival of corresponding messages at the members of a recipient group differ by less than a known upper bound δ.

Figure 2 shows the physical configuration of FTPP. The architecture contains four NEs that are fully connected to each other via fiber-optic links. Each NE hosts up to eight

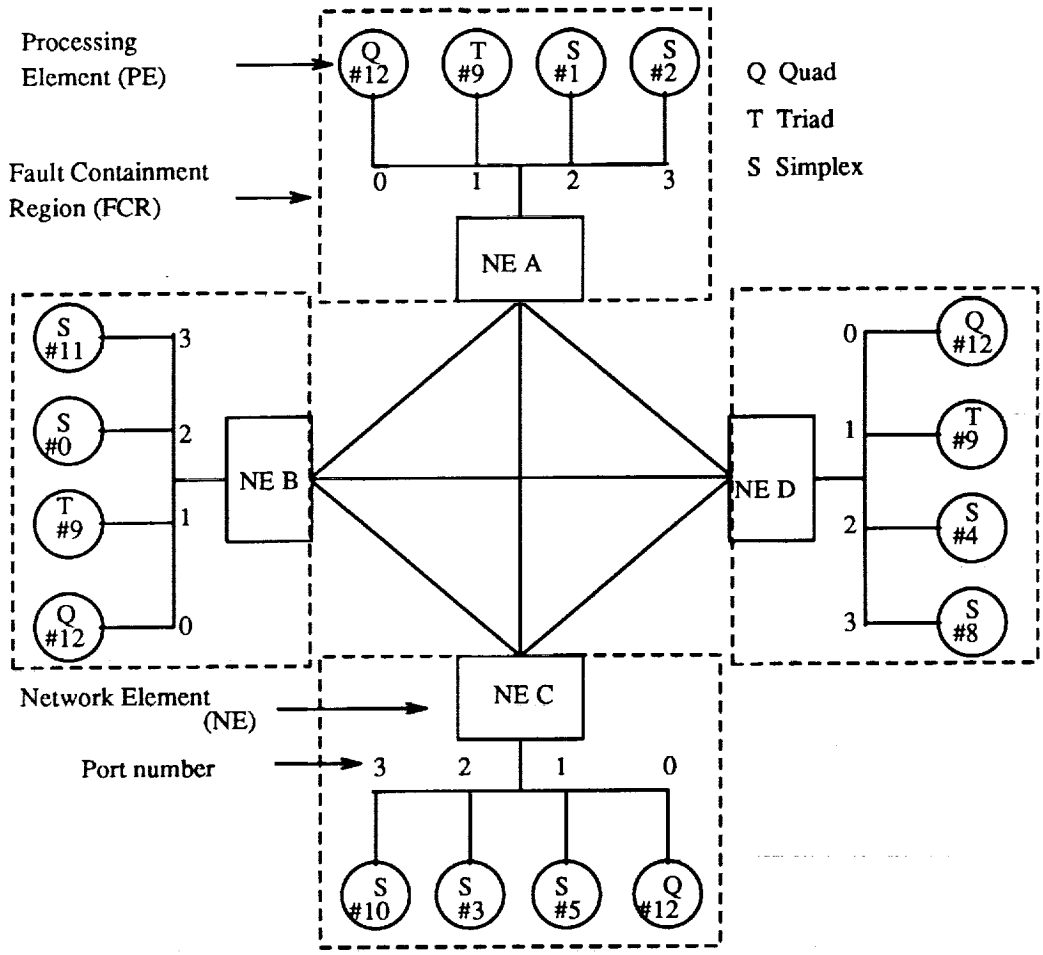


Figure 2: FTPP Physical Configuration

PEs, each of which is a standard processor with a local memory. Each PE communicates with the hosting NE via a standard bus, such as VME. Each NE and the associated PEs comprise a *fault containment region* (FCR) satisfying the requirements for fault containment, namely electrical isolation, physical isolation, independent power, and independent clocking.

The physical configuration is arranged so that the redundant PEs belonging to the same virtual group reside in different FCRs and, hence, are hosted by different NEs. That is, every PE hosted by an NE belongs to a distinct virtual group. Note that an NE does not necessarily host a PE from every virtual group in the system because the redundancy levels of virtual groups can vary between one and four. Every virtual group in the system is assigned a unique identifier called the *virtual group identifier* (VID). Every NE maintains a *configuration table* (CT) that gives the *physical identifier* (PID) of the PEs belonging to every

Port number	0	0	0	0	1	1	1	2	3	3	2	1	2	2	3	3
NE id	A	B	C	D	A	B	D	A	A	D	D	C	B	C	B	C
VID	12				9			1	2	8	4	5	0	3	11	10

Figure 3: FTPP Sample Configuration Table

VID in the system. The PID of a PE is constructed from the unique identifier assigned to the NE to which the PE is connected and the port number through which the PE is connected to the NE. The physical configuration shown in Figure 2 corresponds to the logical organization shown in Figure 1. A sample configuration table that relates the two configurations is shown in Figure 3.

As mentioned earlier, one of the main functions of the NE is to pick up and deliver messages among the members (PEs) of virtual groups. To satisfy the requirements of fault tolerance, the NE hardware itself is replicated among the fault containment regions. To ensure congruent data exchange among the members of a virtual group, the NEs themselves must come to a consensus regarding data originating from and destined to the virtual groups and their members. There are special protocols, called *Byzantine agreement* (or *interactive consistency*) protocols [10], for a set of computing resources to come to a consensus about data private to each of the resources in the presence of Byzantine faults. Each of the NEs employs such a protocol in its implementation of the communication abstraction.

An interactive consistency protocol involves two phases: a message exchange phase, in which the computing resources exchange their private data among themselves, and a voting phase, in which the NEs perform a majority vote on the exchanged copies to arrive at a consensus. Using a majority vote effectively masks the faulty data. The information on which a consensus is sought can be of two basic categories: *group* information or *single source* information. Group information is expected to be identical for every replicated resource. Single source information can be different for each of the resources. For group data, a single round of exchange of data among the resources is sufficient to arrive at a consensus. For

single source data, however, multiple rounds of exchange are required to ensure that every resource accumulates a sufficient number of reliable copies of the single source data associated with every other processor before a majority vote is performed. For example, with four NEs, a single source data consensus requires two rounds of exchanges.

3.2 The Network Element

Operation of the NEs is synchronous and cyclic. A basic cycle, shown in Figure 4, comprises two or more *frames*, where a frame is a period in which all NEs perform a similar (or identical) task. The task performed in a frame may involve all NEs synchronously broadcasting a message to one another.

In the first frame, each NE compiles the message exchange requests originating from all the PEs in the system. This compilation is done in two steps. First, every NE polls each of its own PEs to compile a *Local Exchange Request Pattern* (LERP). Then, the NEs exchange the LERPs among themselves to compile a consistent record of the system-wide exchange requests, namely the *System Exchange Request Pattern* (SERP). Since the LERP is different for each NE, it is necessary to arrive at a Byzantine-resilient consensus on the SERP. Hence, the second step involves a two-round interactive consistency exchange of the LERPs among the NEs.

In the second frame, each NE subjects the SERP to a message approval test to decide which, if any, of the messages must be transmitted. Each NE arrives at the same decision because each executes the same hardwired decision logic on identical input, namely, the consistent SERP. The hardware inside each of the NEs that implements the message approval logic is called the Scoreboard.

In subsequent frames of the cycle, each of the NEs sends all the approved messages to their respective destinations one at a time. The messages exchanged by virtual groups are of two basic categories: *group messages* and *single source messages*. A group message is sent by all members of a redundant group. This category of message is employed only when exact consensus regarding the message to be sent is expected amongst all the nonfaulty members of the sending group. A single source message is originated by a simplex PE or by a single member of a redundant group requiring PE-specific exchange of information, such as a local clock. For either category, the message transmission protocol employed by the NEs ensures

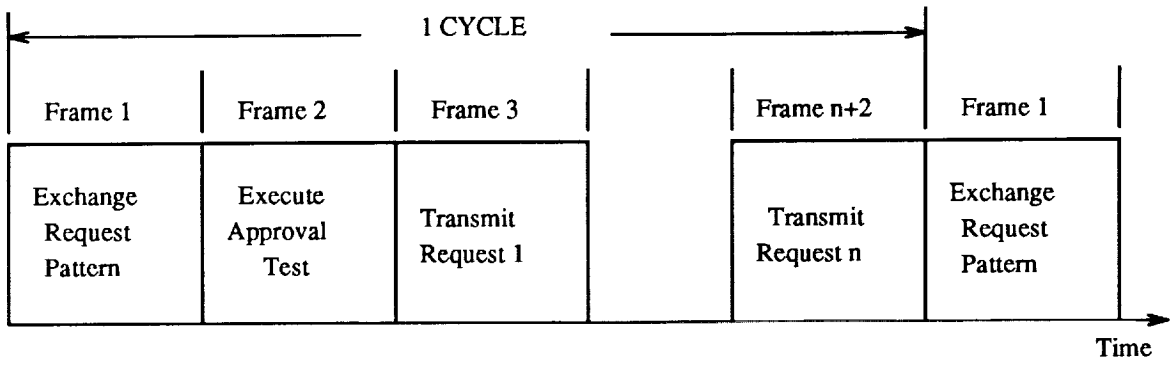


Figure 4: Network Element Cycle

that all the nonfaulty members of the destination group receive *congruent* data, i.e., bitwise identical data. To transmit a group message, each NE exchanges its copy of the message with every other NE, computes a bitwise majority vote on the exchanged copies to create a unique voted copy, and delivers the voted copy to the destination. To transmit a single source message, the NEs perform two rounds of exchange of this message.

3.3 The Scoreboard

A message exchange request for a PE is essentially a record of the status of the *output* and *input buffers* of the PE through which the PE exchanges a message with the NE. The status includes whether the processor has room in its input buffer to receive a message (the input buffer not full, or IBNF, condition) and whether it has a message to send in its out buffer (the output buffer not empty, or OBNE, condition). If the latter is true, the destination VID of the message and a user-defined byte are also included. The user-defined byte includes information regarding the category of the message to be exchanged. The SERP is an aggregate of the buffer status polls of all the PEs in the system.

The Scoreboard uses a virtual-group-to-physical-processor mapping to extract the buffer status information from the SERP on a VG-by-VG basis. Then, it votes the individual buffer status bits contained in the SERP for all members of a VG to determine the overall status of the VG. The Scoreboard approves messages originating from a group only if the following conditions hold.

1. A majority or all the members of the members of the source group are "ready" to send a message. A PE is ready to send a message if its OBNE condition is true.

2. A majority or all the members of the members of the destination group are "ready" to receive a message. A PE is ready to receive a message if its IBNF condition is true.

3. The message satisfies a set of "validity" conditions.

Under some conditions, however, a message from a VG will not be approved even if the above conditions are met. The first exception arises when there is no unanimity on the OBNE (or IBNF) condition for a VID. One of the reasons for lack of unanimity on the OBNE (or IBNF) condition among the members of a group is that some members may be faulty. A possible action that the Scoreboard can take in this case, provided there is at least a majority, is to approve the message and mark the non-agreeing members as faulty. Often, however, the cause for lack of unanimity is that some of the processors are out of synchrony with the others. Note that the PEs belonging to a group, being in different FCRs, are operated by physically distinct clocks. So it is useful to delay the approval of a message, even when there is a majority agreement on the OBNE (or IBNF) conditions for a group, with the goal of tightening the synchrony among the processors. The Scoreboard uses a timeout mechanism to accomplish this goal. When a majority, but not all, of the members are ready to send a message, the Scoreboard starts a timeout for the group, provided a timeout is not already set for the group. The Scoreboard may approve such a message in one of the following NE cycles when the timeout expires or a unanimity is reached.

The Scoreboard makes exception to the approval conditions listed above in two other situations. First, once the Scoreboard approves a message to a destination, it does not approve any additional messages to that destination. The motivation for this decision is that the logic determining the IBNF condition has the logical effect of limiting the length of the input buffers of the PEs to one. Hence, once a message is sent to a PE, its buffer will become full. The second exception arises when dealing with a broadcast message. Since a broadcast message is sent to every virtual group in the system, it can be approved only if the IBNF condition is true for all the groups. A broadcast message takes precedence over ordinary messages. So, once a broadcast message is encountered, no other message will be approved until the broadcast message is approved.

4 Functional Requirements Specifications

Technically, formal verification consists of showing that a certain desired correspondence holds between two formal elaborations—*specification* and *implementation*—describing the same object at two different levels. In the context of hardware verification, the implementation usually describes the hardware design at a certain level of representation. The specification may be a complete description of the expected behavior of the design at a higher level or a description of a set of constraints, such as the desired functional requirements, on the behavior of the design. The value of verification in assuring the correctness of a design critically depends on the correctness of the specification. Some of the qualities desirable in a specification are the following.

- *Abstractness*: A specification must be as implementation independent as possible.
- *Clarity*: A specification must be clear and concise.
- *Completeness*: A specification must describe the requirements as completely as possible.

Our starting point for formulating a specification for the Scoreboard was the master's thesis [9] by Dennis Morton on using the hardware description language VHDL [8] for designing the Scoreboard. The thesis describes the functionality of the Scoreboard using a combination of English and traditional flowcharts. The description given in the thesis is an algorithmic implementation of the Scoreboard. We wanted to construct a more abstract and declarative specification for the Scoreboard. Our requirements specification for the Scoreboard is described below. We formulated an initial version of the specification by going through a process of reverse engineering based on the algorithmic description given in [9] and a description [5] of the context in which the Scoreboard is used. The final version was produced by updating and revising the initial version after a discussion with the CSDL engineering team.

In the following, the Scoreboard is viewed as a black box, shown in Figure 5, that takes commands from the rest of the NE to execute one of a set of possible operations: `clear_timeout`, `Update_CT`, and `process_new_serp`. The internal state of the Scoreboard implementation is hidden in the abstract view except for certain essential parts that are

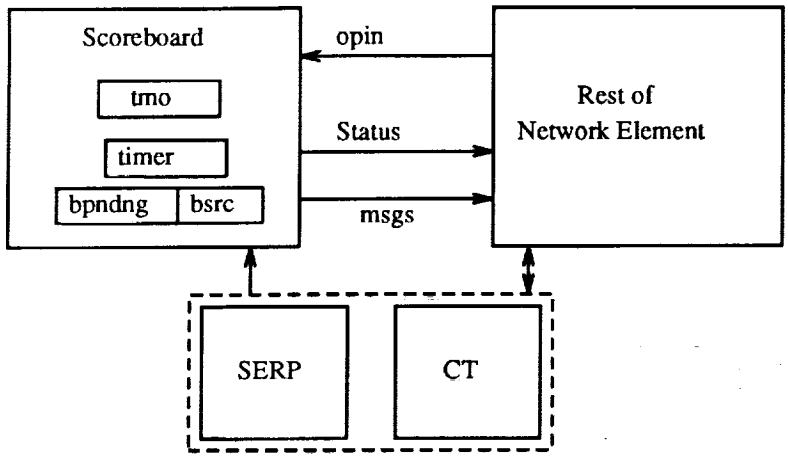


Figure 5: An Abstract View of the Scoreboard

necessary to completely specify the effect of a single invocation of the operations. The Scoreboard state at this level consists of the following items:

- A timeout memory (`tmo`), which records the information necessary to implement the timeout mechanism of the Scoreboard. The `tmo` contains the following information for every valid VID in the system.

  1. Whether a timeout is set for the OBNE condition; if so, the time at which the timeout expires.
  2. Whether a timeout is set for the IBNF condition; if so, the time at which the timeout expires.

- A broadcast pending bit (`bpndng`), which indicates whether the Scoreboard is waiting to approve a broadcast message.

- The source VID (`bsrc`) of the broadcast message, when a broadcast is pending.

- A free running timer (`timer`) that counts the number of clock cycles elapsed during the operation of the Scoreboard; the value of this counter is used to set and check timeouts.

The Scoreboard indicates its internal status during the execution of an operation by means of a signal on the output line `Status`. The signal may take one of the following values: `idle`, `busy`, `message_to_send`. A new operation must be initiated on the Scoreboard only

when `Status` is `idle`. Upon initiation of an operation, `Status` becomes `busy` and reverts to `idle` when the operation is completed.

The operations `clear_timeout` and `Update_CT` reset parts of the internal state of the Scoreboard. The operation `clear_timeout` clears the timeout settings for a VID in the timeout memory `tmo`. `Update_CT` initializes an internal table that the Scoreboard maintains to process the SERP. `Update_CT` must be invoked every time the CT is updated by the rest of the Network Element to keep the internal table consistent with the current state of the CT. In this report, we focus on the specification and verification of processing a new SERP. In our verification, we assume that the Scoreboard is initialized appropriately using the `clear_timeout` and `Update_CT` operations.

`Process_new_serp` is the operation that initiates a new SERP processing cycle on the Scoreboard. This operation produces a sequence of approved messages on the `msgs` line. Every time a new message is produced, the Scoreboard puts out the `message_to_send` signal on the `Status` line. The Scoreboard holds the `msgs` and the `Status` lines constant for at least one cycle and until it receives a `continue` signal on the `opin` line. Upon receiving a `continue`, the Scoreboard starts to compute the next message, which is indicated by `Status` becoming `busy`. Only when all the messages for the current SERP cycle have been sent out does the `Status` line become `idle`.

The most important parts of the behavior of the Scoreboard operation over a single SERP cycle are (1) the list of messages produced by the Scoreboard and (2) the new values for the items in the Scoreboard state. We define these two parts of the Scoreboard behavior at two different levels:

- *Actual behavior*: In terms of a *trace* of the Scoreboard design. A trace is a sequence of values that the state and outputs of the Scoreboard take on over time during a SERP cycle.
- *Expected behavior*: As a function of the inputs (i.e., SERP, CT, etc.) and the initial state of the Scoreboard.

The requirements specification states a desired relation between the actual and expected behaviors. We developed a formal specification of the functional requirements for the Scoreboard in two forms. The *constraints form* (Section 4.3) describes the functional requirements as a set of constraints on the actual behavior of the Scoreboard. The *functional*

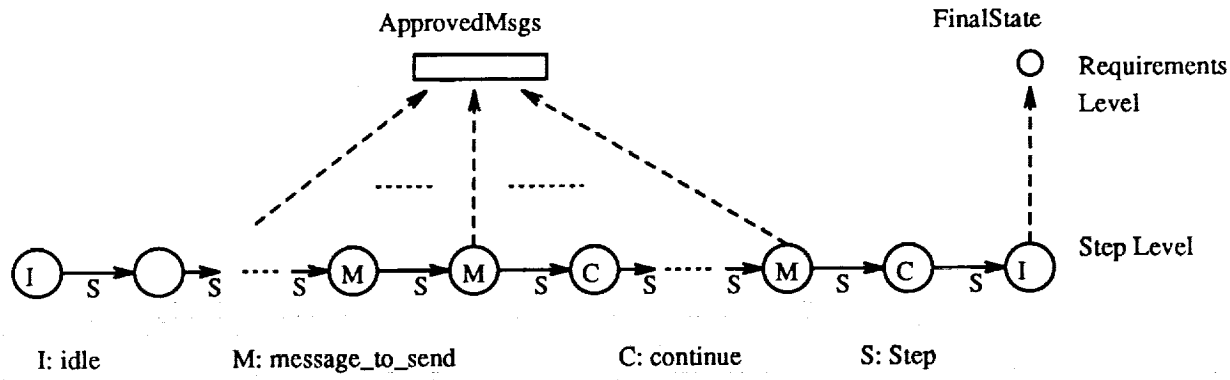


Figure 6: Actual Behavior

form (Section 4.2) defines the expected behavior of the main operation of the Scoreboard by means of a function that maps the initial state of the Scoreboard to its final state and the sequence of all outputs produced during the execution of the operation. Since the behavior of the Scoreboard is inherently procedural, the functional form of specification is involved and not as declarative as one might prefer a requirements specification to be. The constraints form is more declarative and easier to read.

Before we present the requirements specification, we briefly describe how the actual behavior is specified in Caliban to motivate the form in which our requirements specification is expressed.

4.1 Specification of the Actual Behavior

We specified the actual behavior by means of two functions that extract the desired items by generating a Scoreboard trace as shown in Figure 6. `ApprovedMsgs` defines the list of approved messages produced during a SERP cycle, and `FinalState` defines the state of the Scoreboard at the end of a SERP cycle. A Scoreboard trace is generated by a repeated application of the function `Step` to a given initial Scoreboard state. `Step` defines the behavior of the Scoreboard at a certain level of time abstraction that comprises multiple clock cycles. The `Step` level is not our lowest level of specification of the Scoreboard. The `Execute` level (described in Section 5) specifies the structural details and the single clock cycle behavior of the Scoreboard design. Section 6 describes the entire hierarchy of specification levels and how they are related.

Note that a single SERP processing cycle of the Scoreboard involves multiple clock cycles starting and ending in an `idle` state, i.e., a state in which the `Status` output is `idle` (marked `I` in Figure 6) and ends in an `idle` state. A message is picked up by the environment of the Scoreboard every time the Scoreboard moves from a `message_to_send` state (marked `M` in Figure 6) to one in which there is no `message_to_send`.

A Caliban specification of the actual behavior of the Scoreboard is given below. Most of the functions defined below operate on a Scoreboard state `s`, which includes not only the internal state of the Scoreboard but also its inputs, including the CT and SERP, and the outputs. The function `ActualBehavior` defines the actual behavior as a tuple (denoted by `<< >>` in Caliban) of the list of approved messages and the final state. The function `trui` (for trace-until-idle) generates a multi-step trace of the Scoreboard by applying `Step` to an initial state an arbitrary number of times until it encounters an `idle` state. The trace, which is represented as a list in Caliban, will be potentially infinite if the SERP cycle does not terminate. The *lazy* (or *nonstrict*) semantics of Caliban allows one to define such potentially infinite behaviors succinctly and even reason about them. For example, we can prove that a SERP processing cycle terminates by showing that the trace is finite.

`MsgOut` extracts the list of approved messages from the trace by checking every time the Scoreboard makes a transition from a state where there is a `message_to_send` to one where there is no `message_to_send` on the `Status` line. `Final` extracts the internal state of the Scoreboard from the last element of the trace. Both these functions are defined recursively on lists (`[]` denotes the empty list and `(hd:tail)` denotes a nonempty list with `hd` as the head and `tail` as the tail).

```
ActualBehavior s = <<ApprovedMsgs s, FinalState s>>

ApprovedMsgs s = MsgOut (trui (Step s))
FinalState s = Final (trui (Step s))

trui s = [s], is_idle s
         s : trui (Step s)

MsgOut [] = []
MsgOut [s] = []
MsgOut (s1:s2:more) {is_send s1 & ~(is_send s2)} =
```

```
                                        ABS_msgof s1 : MsgOut more
MsgOut (s1:s2:more) = MsgOut (s2:more)

Final [s] = s
Final (a:x) = Final x
```

4.2 Requirements Specification: Functional Form

In the functional form of the requirements specification, the expected behavior is specified as a function (`ExpBehavior`) of the CT, SERP, and the part of the initial state of the Scoreboard that is included in the abstract view of the Scoreboard shown in Figure 5. A definition of this function is described in detail in Section 6.5. Below, we provide a formalization of the requirements specification in functional form in terms of `ExpBehavior`.

The requirements specification is expressed as one or more statements in the Clio assertion language. Any statement that is proved in Clio has to be expressed in the Clio assertion language. The Clio assertion language includes all well-formed sentences from first-order predicate calculus. The atomic literals from which the sentences may be constructed include equations relating two arbitrary terms that may involve functions defined in Caliban. An equation in the Clio assertion language over two Caliban terms `e1` and `e2` is denoted as `'e1' = 'e2'`. The back quotes are used to distinguish the Caliban expressions from the assertion language expressions. A formula in the assertion language may be quantified, existentially or universally, over variables appearing in the Caliban expressions. The default quantification is universal.

```
'ActualBehavior s' = 'ExpBehavior s', Liveness 's'

Liveness 's' :=
   (t) '!(NextContinue t s)' = 'True', '!t'='True'
```

For example, the first sentence given above states that "for all `s` ranging over the Scoreboard state, the actual behavior is equal to the expected behavior provided a precondition defined by the predicate `Liveness` holds for `s`." The comma separates the conclusion of an implication from the precondition. The next line defines the `Liveness` predicate, the

purpose of which is explained below. Although `ExpBehavior` operates on the entire Scoreboard state, the function depends only the CT, SERP, and the parts that are included in the abstract view of the Scoreboard (Figure 5).

For a SERP processing cycle to complete successfully, a `continue` signal must appear a sufficient number of times on the `opin` input line. The predicate `Liveness` ensures this condition. The predicate expresses a constraint on the input stream, which is included as part of the state `s`. The predicate requires that the input stream, which is represented as a function from time to the values of the `opin` input, have an infinite, i.e., an arbitrarily large, number of `continues`. `(NextContinue t s)` returns the earliest instant after time `t` when a `continue` appears in the input stream. The built-in Caliban function `!` is true only if its argument is well defined. Thus, as long as `(NextContinue t s)` is well defined for all time, the input stream is guaranteed to contain an arbitrary number of `continues`.

4.3 Requirements Specification: Constraints Form

A complete definition of the expected behavior as a single function is involved because the Scoreboard behavior is inherently procedural, especially if the time used to set and clear timeouts is based on a free running timer instead of the time at the beginning of a SERP cycle. A definition of the function `ExpBehavior` can be hard to read. In the constraints form, we express the requirements specification as a set of properties that the actual behavior of the Scoreboard must satisfy. A property either expresses a constraint on the actual behavior or relates a piece of the actual behavior to a corresponding piece of the expected behavior. In this style of specification, it is not necessary to define the expected behavior as a single function.

Every property described in the following sections is true only if a set of preconditions is true. Rather than list the preconditions in the statement of every property, we list them below once.

- `InitialCond`: This precondition is a predicate on the initial state of the Scoreboard that must be true for a SERP processing to begin properly. It assumes that the internal state of the Scoreboard is well defined and its internal tables are properly initialized by the resetting operations.
- `Liveness`: This precondition was described earlier. The actual behavior will not cor-

respond to the expected behavior unless the rest of the NE cooperates with the Scoreboard.

- `isvalidvid`: Most of the properties stated below are universally quantified over all possible VID numbers. The quantification over VIDs is actually only over all *valid* VIDs in the system. (A VID is valid if it is the identifier of an existing virtual group.) Hence, every such property is assumed to have a precondition `isvalidvid` over the quantified VID variable.

We classify the properties into five groups, where every group characterizes a certain aspect of the Scoreboard behavior.

4.3.1 Guaranteed Operation Completion

The following properties assert that once an operation is initiated on the Scoreboard it must complete. In the case of `clear_timeout` and `Update_CT`, no external intervention is needed for a successful completion of an operation. For `process_new_serp` to complete successfully, a `continue` signal must appear a sufficient number of times on the `opin` input line. The predicate `Finite` asserts that its trace argument must be finite.

```
Finite 'trui (Step s)', ( 'opinof s' = 'clear_timeout'
                        |'opinof s' = 'Update_CT' )

Finite 'trui (Step s)', ( 'opinof s' = 'process_new_serp'
                        & Liveness 's')
```

4.3.2 Constraints on the Sequence of Messages

The first constraint specifies that the VIDs must be processed by the Scoreboard in increasing order of VIDs. We believe this requirement is not strictly necessary to implement the goal of the network element. It is important, however, that the Scoreboard process the VIDs in some systematic order. We have included the first constraint since it was given to us as a requirement by CSDL, although other orders of processing the VIDS, eg., a round-robin order, might be better to ensure that a message originating from a virtual group is not

ignored arbitrarily long. The second and the third requirements stated below ensure that a broadcast message receives precedence over ordinary messages in getting approved.

1. Messages must be approved in increasing order of source VIDs.

   (In the following, `##` returns the length of a list, `(ithof m msgs)` returns the m^{th} message in the list `msgs`.)

```
IsSrcvidOrdered 'ApprovedMsgs s', Liveness 's'

IsSrcvidOrdered 'msgs' :=
  (n)(m) 'srcvidof (ithof n msgs)
            > srcvidof (ithof m msgs)' = 'True',
               'n > m & n <= ## msgs
                     & m <= ## msgs' = 'True'
```

2. At most one message will be approved, if a broadcast message is pending at the start.

   (In the following, `#1` is a literal that denotes the natural number one.)

```
'(## (ApprovedMsgs s) <= #1' = True',
                'Isbrdcstpndng s' = 'True'
```

3. If there is a broadcast message in the messages approved,
   then it must be the last message approved.

   (In the following, `all` computes the conjunction of all the members of a list of boolean values (returns true, if the list is empty); `map` applies a given function to every element in a given list; and `is_bcst` checks whether a message is a broadcast message.)

```
'all (map (~ is_bcst)
          (AllButLastMsgs (ApprovedMsgs s)))' = 'True',
                          '(## (ApprovedMsgs s)) > #1'
```

4. No two distinct messages among the approved messages may have the same destination.

```
DestnUnique 'ApprovedMsgs s'
DestnUnique 'msgs' :=
  (n)(m) ~('destnvidof (ithof n msgs) =
             destnvidof (ithof m msgs)'), ~('n = m')
```

4.3.3 Message Approval Condition

A message from a source group v may be sent, i.e., may appear in the list of messages produced by the Scoreboard, only if a set of general approval conditions is satisfied by the entries associated with v in the SERP and the initial state of the timeout memory `tmo`. The set of general approval conditions is also sufficient for approval provided the message is not a broadcast and there is no broadcast message pending from the previous cycle. The first constraint given below formalizes this requirement. If there is a broadcast message in the present cycle or there is one pending from the previous cycle, then additional conditions must be satisfied before the message can be approved. The second and third constraints given below formalize the two broadcast processing situations.

First, we formalize the set of general approval conditions by means of the predicate `ApprovalCond`, which is defined as a function of the SERP, CT, and the initial state of the timeout memory `tmo` and `timer`. `ApprovalCond` does not check if a message is valid. The validity condition is included as part of the correctness of the contents of an approved message, described in the next section. Note that even if a message is invalid, the Scoreboard will send the message out, with the destination set to "null," as long as the message satisfies the approval condition.

A message with source VID (v) may be sent only if
the following general approval conditions hold for v:

(a) The voted OBNE for v must be true.

(b) The voted IBNF for the destination of v's message must be true.

(c) The condition under which an OBNE timeout will be set or retained must not hold for v.

(d) The condition under which an IBNF timeout will be set or retained must not hold for the voted destination of v.

The functions `OBNEtimeoutcond` and `IBNFtimeoutcond` define the timeout setting conditions for OBNE and IBNF. Their definitions, which are given Section 4.3.5, are identical except that one applies to OBNE and the other to IBNF. The meanings of the rest of the functions used in the following definition should be self-explanatory.

```
ApprovalCond 'tmo' 'timer' 'vid' 'serp' 'ct' :=
          'OBNEtimeoutcond tmo timer vid serp ct' = 'False'
        & 'IBNFtimeoutcond tmo timer
                (voteddstn vid serp ct) serp ct' = 'False'
        & 'votedobne vid serp ct' = 'True'
        & 'votedibnf
              (voteddstn vid serp ct) vid serp ct' = 'True'
```

1. If no broadcast is pending and a message is not broadcast,
   then a message must be sent iff the message satisfies
   the general approval conditions.

   In the following, the function `bpndng` extracts the `bpndng` part from the initial Scoreboard state `s`; `votedxclass` returns the result of voting the exchange classes in the SERP entries associated with `vid`. The meanings of the rest of the functions should be self-explanatory.

```
(vid)(s)
   ( ((En) 'srcvidof (ithof n (ApprovedMsgs s))' = 'vid')
      <-> ApprovalCond 'tmoof s' 'timerof s' 'vidof s'
                                 'serpof s' 'ctof s' ),
                  (  'bpndng s' = 'False'
                   & 'isbrdcst (votedxclass vid serp ct)' = 'False')
```

2. If a broadcast is pending from the previous cycle,
   then there is a message approved in the present cycle iff
   (a) the source VID of the message is the broadcast
   source stored in the initial Scoreboard state, as well as
   (b) the message satisfies the general approval condition and
   (c) all the valid VIDs are ready to receive.

   In the following, `AllVidsReady` defines the condition under which all the virtual groups in the system are ready to receive a message. The function `brdcstsrcof` extracts the `bsrc` part of the initial Scoreboard state.

```
AllVidsReady 'tmo' 'timer' 'vid' 'serp' 'ct' :=
```

```
(vid)  'votedibnf
         (voteddstn vid serp ct) vid serp ct' = 'True'
     & 'IBNFtimeoutcond tmo timer
         (voteddstn vid serp ct) serp ct' = 'False'


(  'bpndngof s' = 'True'
 & '## (ApprovedMsgs s)' = '#1')
     -> ('srcvidof (ithof #1 (ApprovedMsgs s))'
           = 'brdcstsrcof s')
                <-> ( ApprovalCond 'tmoof s' 'timerof s'
                                   'vidof s' 'serpof s' 'ctof s'
                      & AllVidsReady 'tmoof s' 'timerof s'
                                   'vidof s' 'serpof s' 'ctof s')
```

3. If there is no broadcast pending from the previous cycle,
   and if the message being sourced by a VID is a broadcast,
   then the message may be approved iff the message satisfies
   (a) the general approval condition, as well as
   (b) all valid VIDs are ready to receive and
   (c) this is the first broadcast message encountered.

```
NoEarlierBrdcst 'vid' 'serp' 'ct' :=
   (vid1) ('vid > vid1' = 'True'
             -> 'isbrdcst (votedxclass
                             vid1 serp ct)' = 'False' )


(tmo)(timer)(vid)(serp)(ct)
  ((En) 'srcvidof (ithof n (ApprovedMsgs s))' = 'vid')
    <-> (  ApprovalCond 'tmo' 'timer' 'vid' 'serp' 'ct'
         & AllVidsReady 'tmo' 'timer' 'vid' 'serp' 'ct'
         & NoEarlierBrdcst 'vidof s' 'serpof s' 'ctof s' ),
                      (  'bpndngof s' = 'True'
                       & 'isbrdcst (votedxclass
                             vid1 serp ct)' = 'False' )
```

4.3.4 Correctness of Message Content

For every message that is approved by the Scoreboard, the following conditions must hold:

1. The destination of the message must be the voted destination, unless either the voted destination or the voted exchange class is not valid; in the latter case, the destination is a distinguished "null" value.
2. For every other field of the message, the actual value must be equal to the voted or expected value.

```
ithmsgof i s = ithof i (ApprovedMsgs s)

(vid)(serp)(ct)(n)
   'srcvidof (ithmsgof n s))' = 'vid')
      -> 'votedmsgfor vid serp ct' = 'ithmsgof n s'
```

4.3.5 Correctness of Timeout Setting

In the following, we formalize the condition under which an OBNE or an IBNF timeout must be set or cleared for a VID in the system. The timeout condition is defined as a function of the initial state of the timeout memory `tmo`, the timer, and the SERP and CT entries associated with a VID. We define the timeout condition as a Caliban predicate `OBNEtimeoutcond`, which applies for OBNE. The IBNF timeout condition can be defined in a similar fashion.

For every valid VID,
a timeout must be set for its OBNE
at the end of a SERP cycle iff
the *timeout condition* holds for its OBNE.

```
(vid) 'OBNEtimeoutof vid (tmoof (Finalstate s))'
            = 'OBNEtimeoutcond (srcvidof s) (tmoof s) (timerof s)
                               (serpof s) (ctof s)'
```

Timeout condition for OBNE holds iff

(a) OBNE timeout is not set for the VID at the start and
the *timeout set* condition holds for OBNE or

(b) OBNE timeout is set for the VID at the start and
the *timeout clear* condition does not hold for OBNE.

```
OBNEtimeoutcond vid tmo timer serp ct
   = (OBNEtimeoutof vid tmo)
        -> ~ (OBNEtimeoutclear vid tmo timer serp ct);
             (OBNEtimeoutset vid serp ct)
```

Timeout set condition holds for OBNE iff

the voted OBNE for the VID is true by a majority but not unanimous.

```
OBNEtimeoutset vid serp ct
   = (votedOBNE vid serp ct) & (nonunanmajOBNE vid serp ct)
```

Timeout clear condition holds for OBNE iff

the voted OBNE for the VID is unanimously true or

the voted OBNE for the VID is true by nonunanimous majority and
timeout expired condition holds for OBNE for the VID.

```
OBNEtimeoutclear vid tmo timer serp ct
   =  (votedOBNE vid serp ct)
    & (  (unanOBNE vid serp ct)
       |(  (nonunanmajOBNE vid serp ct)
         & (OBNEexpired tmo timer ct vid)) )
```

Timeout expired condition is true iff

$t_{chk} \geq t_{exp}$, where

t_{chk} is the time when the timeout expiration check is made,

t_{exp} is the expiration value stored in `tmo` for the VID.

```
OBNEexpired tmo timer ct vid
   = (timer + (vidposition vid ct) + 2)
        >= (OBNEexpval vid tmo)
```

The exact value of t_{chk} for a VID in a SERP cycle is implementation-dependent unless, for purposes of timeout calculations, time is considered to be frozen at the start of a SERP cycle. Thus, one consequence of using a free running timer in the design is that a precise characterization of the timeout condition becomes implementation dependent. For our design, t_{chk} is determined precisely by the position of the VID in the VID ordering. If the timer is frozen, then t_{chk} will be the value of `timer` in the initial state.

4.4 Discussion

During the course of the formal analysis, several questions about the Scoreboard functionality arose for which we were not able to get clear answers from the flowchart or English descriptions given in Morton's dissertation [9]. Some of the questions arose during the construction of the functional requirements specification. We clarified most of these questions by discussing the initial version of the requirements with the Draper team. In the following, we list the questions that arose during the verification process, for which we obtained answers from CSDL after the verification was completed. Apparently, some of the questions helped the CSDL team to shed new light on certain aspects of their design.

Question 1: Should the validity and approval condition of a *pending* broadcast message that is sent out in a SERP cycle be checked and ensured explicitly by the Scoreboard?

In other words, can the validity of a pending broadcast message be assumed implicitly by imposing a suitable precondition on the environment? The precondition to be imposed is that the SERP entries associated with the source VID of the broadcast must not change between two broadcast pending SERP cycles. If the Scoreboard does not have to guarantee the validity on its own, then Property 2 in the message approval condition part of the requirements specification (Section 4.3.3) can be weakened by including a precondition on the current SERP. If the Scoreboard has to guarantee the validity even if the SERP can change, then the Scoreboard must save the entire broadcast message for future use. We were told by CSDL that the Scoreboard may assume that the pending message remains valid across SERP cycles so the design does not have to explicitly check for this condition.

Question 2: Should an invalid broadcast message be sent?

When an ordinary message is invalid, the Scoreboard sends the message anyway, but only after setting the destination of the message to "null." The question is, should the same be done for a broadcast message? The message content correctness property (Section 4.3.4) assumes that such "null" broadcast messages are sent. The CSDL team agreed that this was a relevant question and that there should be a message produced in such a situation.

Question 3: Can more than one message be sent to the "null" destination?

Based on the requirement that every invalid message that is approved must be sent to a "null" destination, it appears that there have to be multiple messages with "null" destination in the output. The requirement that no two distinct messages may have the same destination, which, presumably, does not include the "null" destination, applies only to valid VIDs in the system. The CSDL team clarified that the Scoreboard should indeed produce multiple "null" messages, although they are going to be discarded, because the existence of a message gives useful information to the NE.

Question 4: Should the Scoreboard use a free running timer to set and check its timeouts or just use the time at the beginning of the SERP cycle?

As we discussed earlier, the advantage of freezing the time for timeout calculation purposes is that it enables developing a complete functional characterization of the behavior (i.e., the sequence of messages approved) of a single SERP cycle in a design-independent fashion. If a free running timer is used, the definition of the timeout expiration condition (`OBNEexpired`) used in the requirements specification becomes specific to a particular design.

Initially, we were apprehensive that this would complicate the specification and verification significantly. As it turned out, at least for our high-level design, this was not the case. We were able to define the time at which a timeout is checked for a particular VID without too much difficulty. For a more detailed hardware design, however, it may be harder to characterize the timeout condition as a function of the initial state, especially if the relation between the master clock and the free running timer is more complicated than it is in our high-level design. In that case, we may have to formulate and verify a weaker version of the requirements specification for the timeout condition.

5 A Scoreboard Design

Our Scoreboard design is based primarily on the flowchart and VHDL descriptions of the Scoreboard given in Morton's thesis [9]. We started by structuring our design close to the flowchart description, for our goal was only to construct a high-level design. The eventual design, however, turned out to be closer to the VHDL design in its level of detail than the flowchart because the flowchart had to be refined to obtain a fully operational design. The Scoreboard interface corresponds closely with that of the VHDL design. Our design implements more of the Scoreboard functionality than does the VHDL design. For example, our design processes broadcast messages in the SERP; the flowchart and the VHDL design do not. We implemented this feature by consulting the CSDL personnel about the requirements for broadcast.

We constructed and specified our design using Spectool. Excluding three main blocks, all the components used in the design are either ordinary registers or special kinds of registers. The design is detailed enough that it can be translated into register transfer level hardware by refining the main blocks used in the design one level further. Some of the signals on the wires and the internal states of the components are represented using high-level data types, such as records and natural numbers. Obtaining a hardware realization for the Scoreboard was not one of our goals in Phase 1. Hence, we did not use hardware efficiency as the main criterion in choosing our design decomposition. Before describing our design, we give a summary of our hardware model.

5.1 The Hardware Design Model

A finite state controller circuit consists of a *data path* and a *controller*. The data path is a set of interconnected components. Some of the components can be connected to input ports (represented by diamonds) and output ports (represented by circles) for external communication. All components in a class share the same set of structural and behavioral *attributes*. The structural attributes include the number of input/output ports, the type of signals expected at the ports, the icon used to denote a component in the class, etc. The main behavioral attribute of a component is the set of *actions* that can be triggered by the controller. An action, when triggered on a component, may cause a change in the internal state and/or the state of the signals at the outputs of a component. The effect of an action is

defined as a function of the current state and current inputs of the component. For example, an arithmetic logic unit (`ALU`) can be defined by having an action for every arithmetic/logical operation it supports.

The controller is a finite state machine that gets inputs from the data path components or from outside the circuit. A controller state transition corresponds to advancement of a unit of discrete time (a *cycle*) on an implicit global clock. In every state, the controller sends control signals to the data path that cause a set of actions to be triggered on the components. The set of actions triggered in a controller state is called the *schedule* (of actions) for that state. In our circuit representation the control signals are not shown explicitly. A schedule is specified by giving the symbolic names of the actions triggered by the control signals. In addition to scheduling actions on components, a controller may send outputs outside the circuit. These external outputs from a controller are shown explicitly in the circuit diagram.

A clock cycle may be further divided, at the user's discretion, into a finite number of *phases*. A phase is the smallest unit of time used in a circuit specification. The schedule for a state is really a chronological sequence of *schedule fragments*, one for every phase in the state. A schedule fragment is defined as a function of the current controller state, phase, and inputs. The actions in a schedule fragment associated with a state-phase pair are triggered simultaneously. Every action defined on a component has an associated delay, which is measured in integral number of phases. The results of an action with delay `delta` triggered in a phase `phi` are available in phase `(phi+delta)`. A state transition always takes place in the last phase of a cycle.

5.2 Informal Description

The Scoreboard design goes through two main loops—*voting* and *checking*—in performing a SERP cycle. In the voting loop, it iterates over all the valid VIDs in the system, collects all the SERP entries belonging to the same virtual group, votes them, and stores them in an internal voted SERP memory (`VSERP`). It also checks and sets or clears, as necessary, timeouts for the IBNF and OBNE conditions for every virtual group in this loop.

In the checking loop, the Scoreboard cycles through the `VSERP` to check if there are any `VSERP` entries from which a message that satisfies the message approval condition can be generated. If and when it finds one, it enters the *sending* loop to send the approved

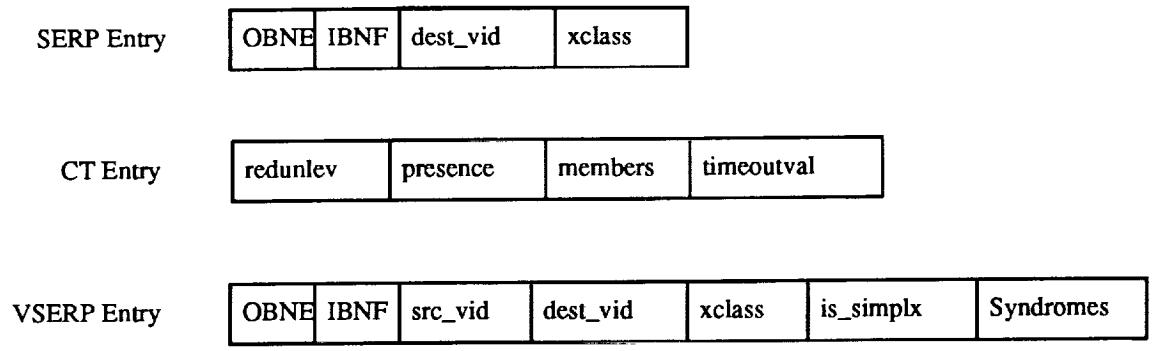


Figure 7: The SERP, CT, and VSERP Entries

message and waits for a `continue` signal for proceeding further in the checking loop. When it encounters a broadcast message, it enters the *broadcast* loop to check if the broadcast message can be approved and sent. If a broadcast message is pending approval from the previous SERP cycle, the Scoreboard enters the broadcast loop from the voting loop, skipping the checking loop entirely, because approval of a broadcast message must take precedence over sending all other messages.

The entries in the SERP memory, `SERP`, are indexed by the processor identifiers (PID). The fields of a typical SERP entry are shown in Figure 7. The VIDs, which are represented as natural numbers (`NAT`) in our design, are assumed to be bounded by a maximum, `max_vid`, which is treated as an unspecified constant in our specification. Not every VID may be in use at any given time; the VIDs that are in use are said to be *valid*. The configuration table, `CT`, has an entry for every possible VID number.

The fields of a typical CT entry are shown in Figure 7. A CT entry for a VID gives the redundancy level, the PIDs of the member PEs belonging to the virtual group denoted by the VID, and the offset value to be used for setting timeouts for the VID. The entry also contains a `presence` field that indicates, for every NE, if the VID has a member subscribed to the NE. An invalid VID is given a redundancy level of zero. The Scoreboard uses the CT to fetch entries belonging to the same group before voting and uses an internal lookup table, `LKUP`, to skip entries in CT that correspond to invalid VIDs. The lookup table, which is generated during initialization by the `CT_Update` operation, also stores the actual number of VIDs (`numvid`) in the system, and performs the termination check required for the voting and checking loops.

Figure 7 shows the fields of a typical VSERP entry. The values of most of the fields

in the VSERP entry for a VID are the results of voting the corresponding fields from the SERP entries belonging to the VID group. The only exceptions are for the OBNE field, the IBNF field, and the new fields Syndromes and src_vid. The OBNE (or IBNF) field contains its corresponding voted result, unless a timeout is set in the voting loop. If a timeout is set, the field is cleared irrespective of the value of the voted result. This kind of setting eliminates the need, outside the voting loop, to access the timeout memory to check if a timeout is set on a VID.

The Syndromes field is a piece of information that is generated in addition to the voted result. It indicates, for every voted field, if the actual value of the field for a member PE belonging to the VID group differs from or agrees with the voted result. This information is included in the message delivered to the NE. The NE uses the syndrome information to identify the faulty PEs and take appropriate actions.

5.3 The Scoreboard Data Path

Figure 8 shows the data path diagram of our Scoreboard design. Although the SERP and CT are shown in the data path, they are not part of the Scoreboard design. They are included in the design specification for convenience; we could have used instead the data lines from the SERP and CT as inputs to the Scoreboard. The address input (coming out of the CT) to SERP is a tuple consisting of a redundancy level number and four PIDs. The output of the SERP is a list of SERP entries. When a read is performed on the SERP, the SERP uses the redundancy level information in the input to output a list of entries of an appropriate length.

The major component blocks of the Scoreboard are LKUP, VOTE, TMO, VSERP and VALID. The rest of the components are multiplexers (MX and MX2), ordinary registers of different sizes (VS, MSG, and OBNE) and special kinds of registers (BOK and BDCST). The Scoreboard also contains a free running timer (TIMER) that is incremented once every clock cycle; the TIMER can also be initialized. The table in Appendix B lists all the actions, along with their delays, defined on each of the components used in the design. Next, we describe some of the major blocks used in the design.

LKUP implements a lookup table with some special-purpose logic. The lookup table is used to generate the set of all valid VID numbers in the system in sequence and in increasing

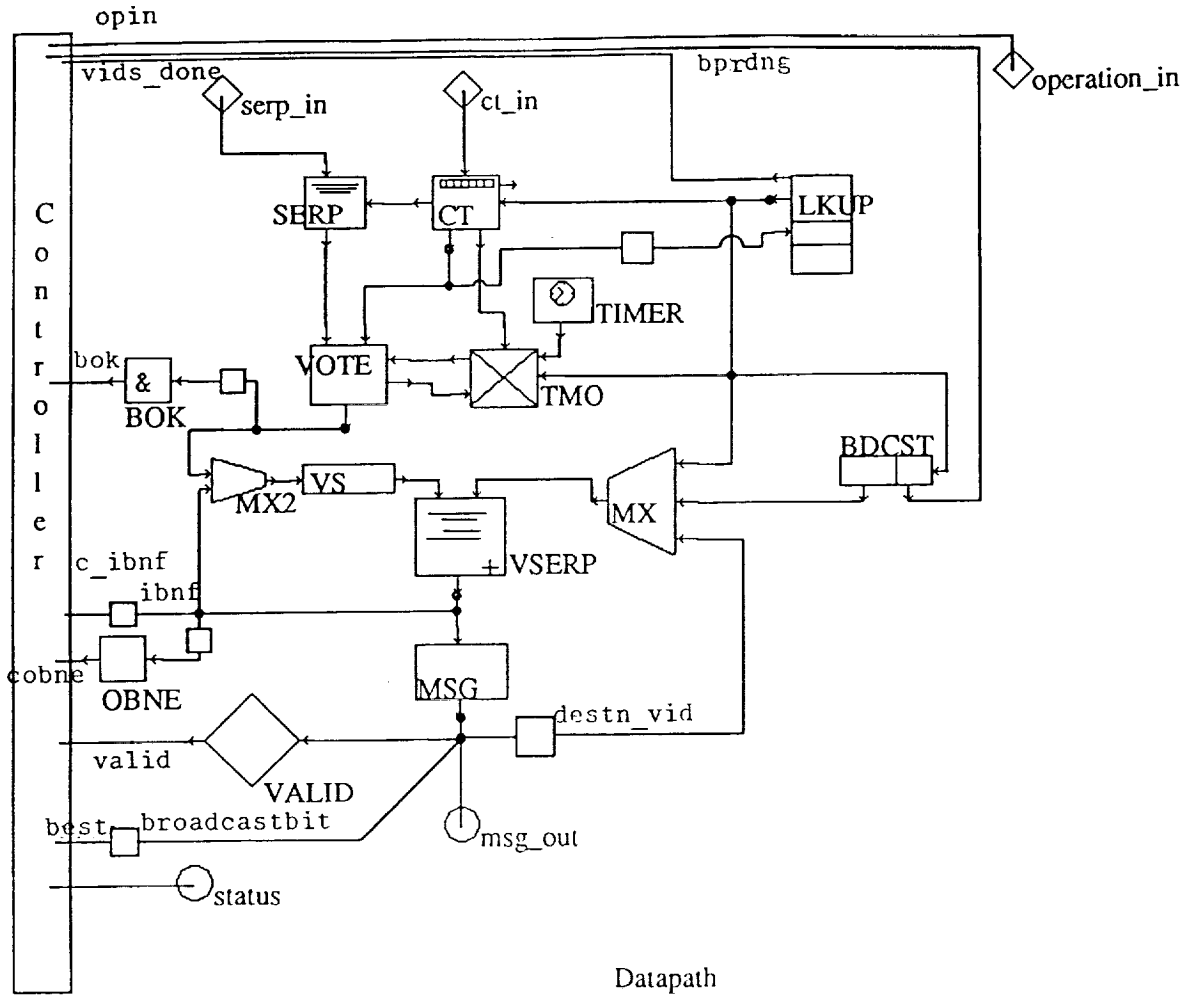


Figure 8: The Scoreboard Data Path

order. This table is pre-compiled and stored for a particular CT during the CT_Update operation of the Scoreboard. The actions defined for LKUP are reset_index, next_index, set_table, and reset_table. The only action of LKUP that is invoked during a SERP processing cycle is next_index. Performing next_index causes the next valid VID to appear on one of the outputs of the block (the one that is input to CT). After the largest valid VID has been generated, next_index causes LKUP to cycle back to the smallest VID. This action also generates another 1-bit output (the one labeled vids_done that is routed into the controller), which indicates whether the current VID generated was the largest valid VID in the system. Note that vids_done can be used to check whether one has cycled through all the valid VIDs in the system once.

The inputs to the VOTE block are the SERP entries (from SERP), the redundancy level of the VID (from CT), and the current status of the timeout settings (from TMO) for the OBNE and the IBNF conditions for the VID. The block has a single action vote defined on it. Performing this action causes the following to be output: the voted result (coming out on the bottom output of the block indicated in Figure 8) and new timeout settings (set, clear, or retain coming out on the side output of the block indicated in Figure 8) for the IBNF and OBNE conditions.

TMO is a timeout memory that has an entry for every valid VID. Every entry contains two fields, one for OBNE and one for IBNF. Each field contains a pair of items indicating whether a timeout has been set, and if so, the absolute time at which the timeout would expire. The inputs to the block are the current time (from TIMER), the VID index (from LKUP), the offset value (from CT) to be used to set the timeout for the VID, and signals giving timeout setting instructions (from VOTE). The actions defined on TMO are reset_tm, get, and set. The action get outputs the current status of the timeout settings for the OBNE and the IBNF conditions after checking for expiration; set sets the timeouts for the current VID based on the instructions supplied at the input; and reset_tm resets every entry in TMO.

BDCST is a two-field register with the following actions: set_bp and clear_bp. One of the fields is a 1-bit quantity, which is used to indicate if a broadcast is pending from a previous cycle. The other field is used to save the VID number of the group that is sourcing the broadcast message pending transmission. The action set_bp sets the VID field to the value on the only input to the block and also sets the 1-bit field to true; clear_bp clears the 1-bit field.

The special-purpose 1-bit latch BOK is used to compute and store the conjunction of the IBNF conditions of all the valid VIDs in the system. Its input is the IBNF field of the voted result coming out of VOTE. The action **and_bok** stores the conjunction of the current input and the current state of the block.

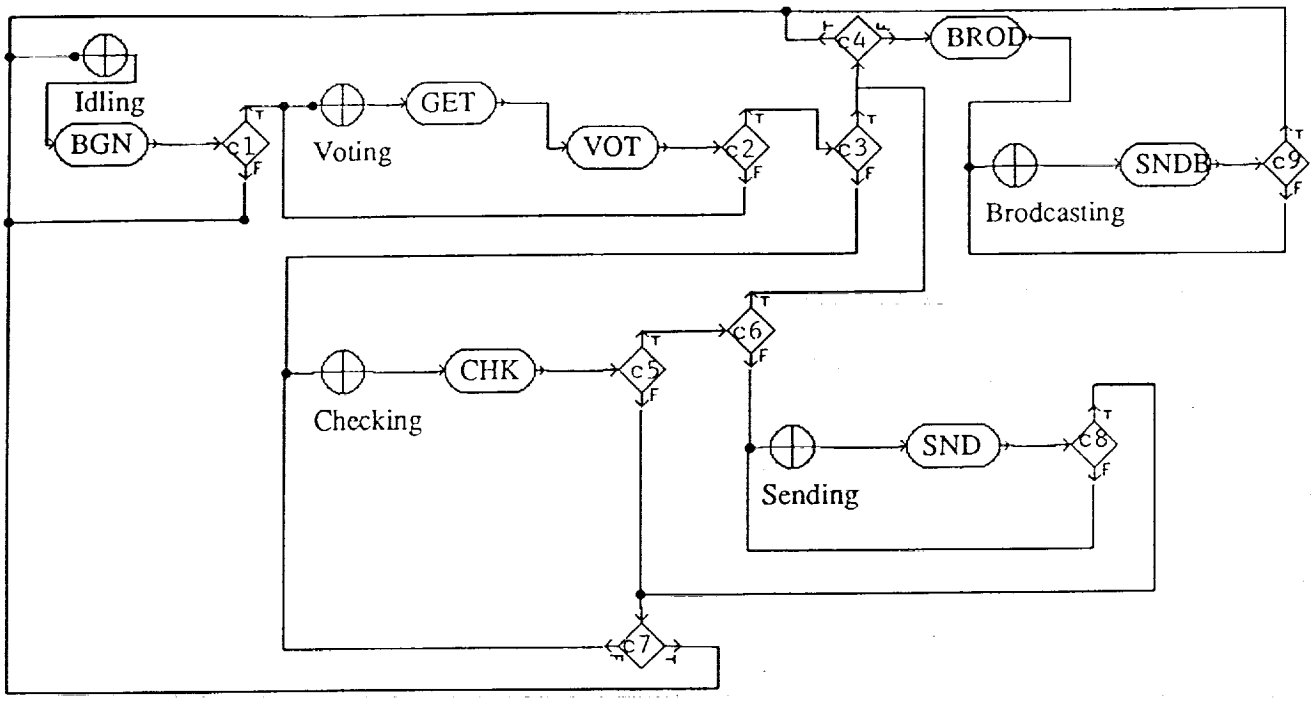
5.4 The Scoreboard Controller

Figure 9 shows the finite state machine implemented by the Scoreboard controller. The ovals denote the controller states and the diamonds denote conditional branches. The boolean expression attached with each of the diamonds is given in the table that is included at the end of Figure 9. The boolean expressions are functions of the inputs to the controller, which are shown in Figure 8. The special symbol $\oplus$ marks selected points of control, the beginning of all the loops in the present case, in the controller. These marked points facilitate the generation of some of the *verification conditions* needed in our proof of correctness.

In the BGN state, the controller idles, waiting for a command to execute an operation. Upon receiving a `process_new_serp` command, the controller moves to GET to begin a new SERP cycle. The voting loop involves the states GET and VOT. The checking loop starts in CHK. The controller enters SND state only if the current message is approved for sending. The controller skips the checking loop entirely and enters the BROD state if a broadcast is pending from the previous cycle (i.e., bpndng is true) and the IBNF conditions are true for all VIDs (i.e., bok is true). BROD can also be entered from inside the checking loop if an approved broadcast message is encountered in the current SERP cycle and the bok condition is true. SNDB is the state in which a broadcast message is sent. Note that in both SND and SNDB the controller idles until a `continue` is received.

The controller clock cycle is divided into five phases, with the last phase reserved for advancing the controller state. Thus, in every state, the controller has four phases to schedule actions on the data path components. The TIMER is advanced exactly once per cycle in every state in phase 3. The complete schedule of actions performed by the controller is given in Appendix A. In the following, we summarize the overall effect of all the actions performed in each of the controller states.

GET: Fetch SERP entries belonging to the current VID by reading the CT and SERP in sequence; get the OBNE and IBNF timeout status for the current VID from TMO.



Controller State Machine

```
C1 (opin = process_new_serp)
C2 vids_done
C3 bpndng
C4 ~(bok)
C5 c_obne & ((bcst & valid) | (~bcst & c_ibnf))
C6 bcst
C7 vids_done
C8 (opin = continue)
C9 (opin = continue)
```

Figure 9: The Scoreboard Controller

VOT: Perform `vote` action on `VOTE` (note that this action not only votes the SERP entries, but also determines if IBNF and OBNE timeouts need to be set, retained, or cleared for the VID and generates syndromes); set or clear, as necessary, OBNE and IBNF timeouts in `TMO`; latch the voted SERP entry into `VS`; perform an `and_bok` action on `BOK` with the voted OBNE; enter the voted result into `VSERP`; advance the `LKUP` index.

CHK: Fetch the next entry from the `VSERP`; save the source OBNE in the latch `OBNE` for later use; latch the message contents from the VSERP entry fetched into `MSG`; perform `check_validity` on `VALID` to check the validity of the current message; fetch VSERP entry of destination VID (with destination VID routed via the multiplexer `MX`); latch the VSERP entry of the destination VID into `VS`; if the message is invalid, then set the destination of message to "null"; if the current message is broadcast and valid, then, if necessary, set the broadcast pending and source fields of `BDCST`.

SND: Clear the IBNF bit of the VSERP entry corresponding to the destination VID so that no more messages will be approved for this destination (this is done by clearing the `IBNF` field in `VS`, which has a copy of the VSERP entry for the destination VID, and entering the modified content of `VS` into `VSERP`).

BROD: Clear the broadcast pending field of `BDCST`; read `VSERP` at the VID contained in the source field of `BDCST`, which is routed via the multiplexer `MX`; generate the message and latch it into `MSG`.

SNDB: No action other than the default actions that are performed in every state.

5.5 Formal Design Specification

The behavior of a finite state controller circuit is uniquely determined given the following:

- The data path and controller structure.
- The controller schedule.
- A specification of the effect of the actions on every component.

The specification of a design (generated by Spectool) consists of the following parts. The first three parts give the Caliban specification of the information listed above.

1. *Structural* specification describes the structural aspects of the data path: the names of the data path components, the external inputs and outputs of the circuit, and the connections between components.

2. *Controller* specification describes the controller finite state machine.

3. *Component classes* specification gives a description of each of the component classes used in the data path.

4. *Composite behavior* specification defines a set of Caliban types and functions that derive the behavior of a circuit from the information specified in the rest of the specification.

The complete details of the structure of a design specification can be found in [14]. In the following, we summarize the top level of the composite behavior part of the specification.

The two main top-level functions defined by the design specification are `Execute` and `Output`. `Execute` advances the state of the system across a single cycle. `Output` returns the (tuple of) external outputs produced by the circuit over the next cycle. `Output` actually returns a list of outputs, one for every phase in a cycle that has been designated as an "output phase." The two functions are defined hierarchically in terms of several other functions, some of which are described below.

The type `STATE` defines the data path state as a tuple of three components. `SYSTEM_STATE` maps every component (an element of type `COMP`) in the data path to its `LOCAL_STATE`. The type `[CHANGE]` is a list of *update tuples* that keeps a record of all the *pending* actions on components, i.e., actions that have been triggered by the controller, but are yet to be completed. This list is maintained to simulate the effect of delays on actions.

The first two fields of `STATE` define a snapshot in time of the data path state. The state of a component is given by `SYSTEM_STATE` unless an action is pending on the component, in which case the state is `bottom`.

The third field of the tuple, `INPUT_STREAM`, specifies the values of the external inputs to the circuit as a function of time. `INPUT_STREAM` is a function type from time (`NAT`) to `EXTSTATE`, where `EXTSTATE` is a type that combines all the external inputs into a tuple.

The type `LOCAL_STATE` is a labeled union of the *local states* of all the components of the data path (and the controller). The local state of a component is a tuple of its internal

state and (a tuple of) its outputs. The definition of LOCAL_STATE appears in the component classes specification.

The function update_state causes the result of all the update records on the list that have timed out to take effect on the circuit state. The function also decrements the time out counter of the update records that have not yet timed out by one unit of time. The function update_state uses do_changes to accomplish the desired goal. The purposes of the rest of the functions are summarized below.

- do_phases updates the state for all the phases in a cycle.
- do_phase updates the state for a single phase. The function causes (using update_state) the pending updates that have timed out to take effect; gets the current_schedule from the controller; makes new update records (using do_actions) for all the actions in the schedule and adds them to the state; and then advances the input stream by a time unit.

```
type STATE = <<SYSTEM_STATE, [CHANGE], INPUT_STREAM>>
type SYSTEM_STATE = COMP -> LOCAL_STATE
type INPUT_STREAM = NAT -> EXTSTATE
CHANGE ::= Change !COMP !NAT LOCAL_STATE

Execute :: STATE -> STATE
Execute s = do_phases 0 s


do_phases :: NAT -> STATE -> STATE
do_phases n s = update_state s , n = num_phases
                do_phases (n+1) (do_phase n s)

Output :: STATE -> Output_type
Output s = generate_output 0 s


generate_output n s {n=num_phases} = []
generate_output n s {output_phase n} =
  Out(do_phase n s) : generate_output (n+1) (do_phase n s)
```

```
generate_output n s = generate_output (n+1) (do_phase n s)

Out s = current_outs (state_function s)(inputstream s)

||clio: modify_rule "do_phases" count 20000
||clio: symbol do_phases never

update_state <<s,p,in>> = do_changes p <<s,[],in>>

do_phase :: NAT -> STATE -> STATE
do_phase n <<s,p,in>> =
  advance_inputstream (do_actions (current_schedule s2 n) s2)
  where s2 = update_state <<s,p,in>>

current_schedule s n = scheduler (controllerstate s) (controllerinput s) n
```

6 The Correctness Theorems

As is common in most large verification efforts, we decomposed the verification of the Scoreboard design into a hierarchy of smaller and more manageable lemmas. To facilitate the decomposition process, we introduced two additional layers of specification between the Spectool generated specification (execute level) and the expected behavior specification (requirements level) for the Scoreboard. In this section, we briefly describe the intermediate levels and the some of the important lemmas that had to be proved to relate the levels in the specification hierarchy. Figure 10 illustrates the specification and the lemma hierarchy used in the verification process.

6.1 The Execute Level

At this level, the Scoreboard is specified by means of the function **Execute**, which defines a single clock cycle behavior of the Scoreboard design, i.e., the behavior for a single transition of the Scoreboard controller. This level of specification was described in detail in Section 5.

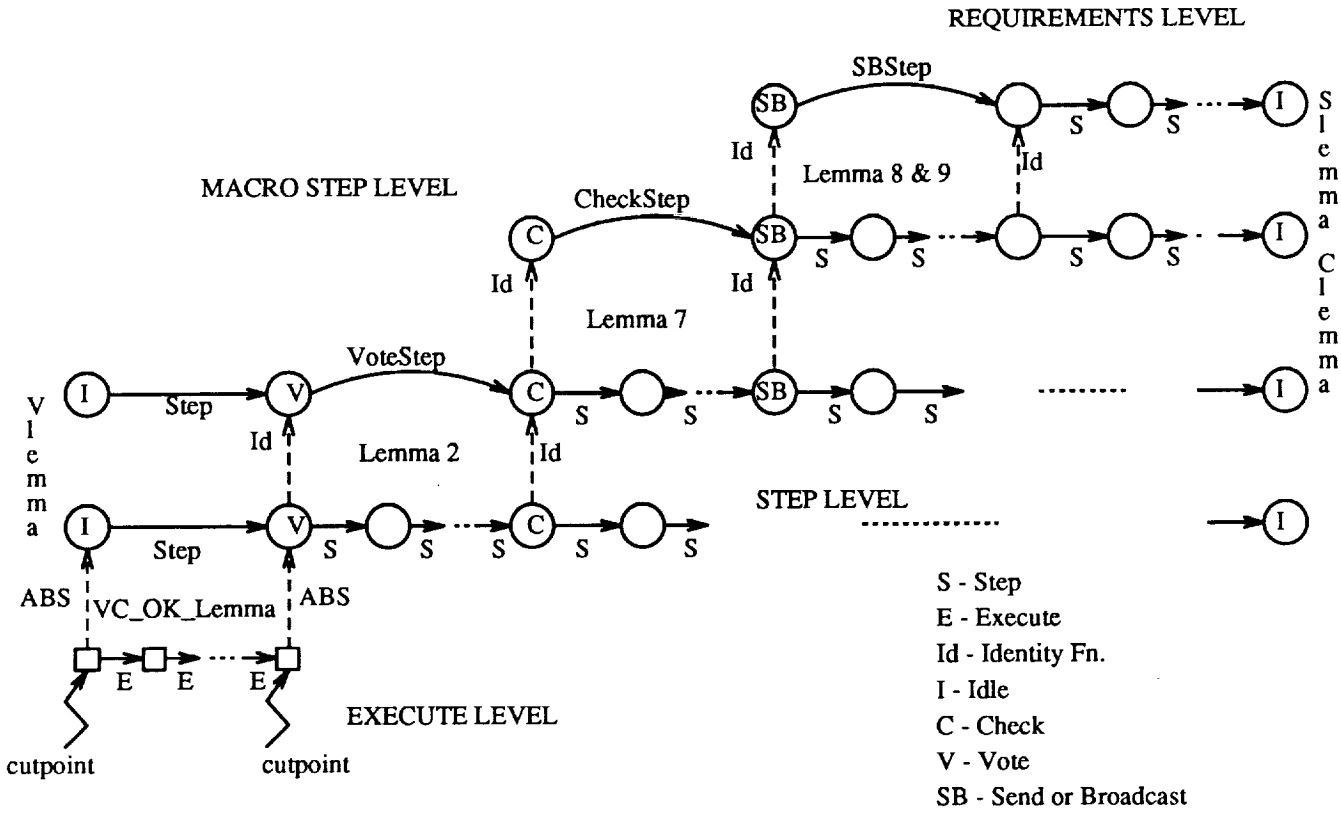


Figure 10: The Lemma Hierarchy

6.2 The Step Level

At the next higher level, the Scoreboard is specified by the function `Step`, which defines the behavior of the Scoreboard between two successive cutpoints ($\oplus$) in the controller. That is, `Step` defines the behavior of the circuit for a set of "kernel" paths, which consists of all paths between pairs of successive cutpoints (not necessarily distinct) in the controller. The behavior for all paths that the controller would traverse during a SERP cycle can be composed from the behavior over these kernel paths because we have placed enough cutpoints to cut every loop in the controller.

The Scoreboard state at the step level is an abstraction of the state at the execute level. For example, the signals on the internal wires, which are included in the state at the execute level, are abstracted at the step level. The function `ABS` relates the states at the two levels.

`VC_ok_lemma` formalizes the correctness relation, denoted by the box labeled `VC_ok_lemma` in Figure 10, between the step and execute levels. The `VC_ok_lemma` states that, for each execution path (from cutpoint to cutpoint in the control state diagram), the effect of iterating over an appropriate number of **`Executes`** is equivalent (under the mapping `ABS`) to performing a single `Step` function.

6.3 The Macro Step Level

The next higher level, called the macro step level, defines three functions that specify the effect of the four major blocks of computation that take place during an arbitrary SERP cycle.

1. `VoteStep` defines the effect of the voting loop, including the first step taken from the idle state, i.e., the combined effect of the computation that starts from `BGN`, then enters the voting loop, and proceeds all the way to the instant the Scoreboard exits the voting loop.

2. `CheckStep` defines the effect of the Scoreboard computation from the instant the Scoreboard is in `CHK` state to the next instant at which it is in one of the sending states (`SND` or `SNDB`) or back in `BGN`. That is, `CheckStep` abstracts the loop from `CHK` to itself.

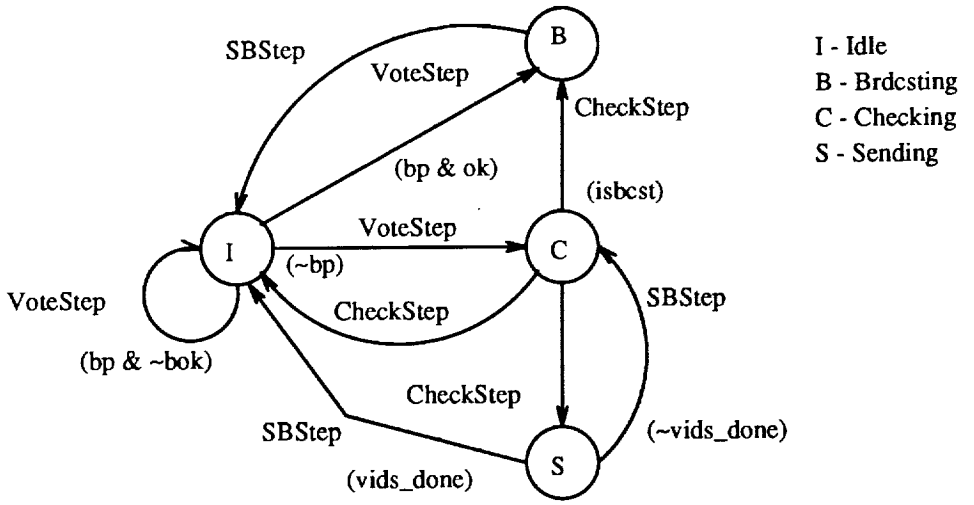


Figure 11: Computation in Macro Steps

3. `SBStep` defines the effect of the two sending loops in the Scoreboard in a single function. That is, it defines the effect of the Scoreboard computation from the instant the Scoreboard is in either `SND` or `SNDB` state and the next moment that it is out of these states.

Figure 11 shows the abstract control diagram for a SERP cycle computation in terms of the macro step functions listed above. In the figure, the abstract controller states are labeled `I` for idle, `C` for validity checking, `S` for normal sending, and `B` for broadcast sending. The condition under which an arc is traversed, in the case of a multiway branch from a state, is included as a label for the arc.

Note that every macro step function must have the same effect as executing a certain number of `Steps`, where the number of `Steps` can be defined as a function of the Scoreboard state to which the macro step function is applied. For example, the number of `Steps` that are required to accomplish the effect of a single `VoteStep` is equal to the number of valid VIDs in the system. We proved four lemmas (`Lemma 2`, `Lemma 7`, `Lemma 8`, and `Lemma 9`) to relate the `Step` and macro step levels. A formal definition of each of these lemmas is given below and the relationship that they establish is illustrated in Figure 10.

```
VoteStep s = VotingStep (numvidsof (ABS_lkupof s)) s
Lemma2 := (i::NAT)(n::NAT)
            'iterate (Succ i) Step s' = 'VotingStep i s',
                    StartSerp 's' 'Succ n' & 'i <= Succ n'='True'
```

```
Lemma7 := 'iterate (checksteps s) Step s' =
                  'CheckStep s', NormalLkup '(ABS_lkupof s)'

SendStep s = Sendingstep (sendingsteps s) s
Lemma8 := 'iterate i Step s'='SendingStep i s',
                              'i <= (sendingsteps s)'='True'

BcstStep s = BcstingStep (bcstingsteps s) s
Lemma9 := 'iterate i Step s'='BcstingStep i s',
                                   'i <= (bcstingsteps s)'='True'
```

Lemma 2 is stated in terms of (VotingStep i s), which is equivalent to (VoteStep s) when i is the total number of VIDs in the system. The lemma asserts that for every i less than or equal to the number of VIDs the effect of iteration in terms of Steps is equivalent to the effect of applying (VotingStep i). The precondition StartSerp ensures that the state s is initialized properly to start a new SERP cycle; it also ensures that (n+1) is equal to the number of VIDs. Our design requires that the number of VIDs in the system be at least one.

In Lemma 7, which states a similar requirement as Lemma 2 for CheckStep, checksteps defines the number of Steps required. The precondition NormalLkup ensures that the state s is initialized appropriately for a proper execution of CheckStep.

Lemma 8 and Lemma 9 together state the corresponding requirement for SBStep. SBStep is itself defined in terms of SendStep or BcstStep depending on whether the message being sent is a broadcast or not. Hence, there are two separate lemmas for SBStep. The number of steps that have to be performed is defined by the functions sendingsteps and bcstingsteps, respectively.

6.4 The Requirements Specification Level

The topmost level in the hierarchy is the level at which the requirements specification (functional form) was expressed. As we described in Section 4.2, the main function that specifies the Scoreboard behavior at this level is ExpBehavior. We describe the definition of

ExpBehavior next in Section 6.5. First, we describe the main lemma that we proved for the Scoreboard.

As we described in Section 4.2, the assertion that completely formalizes the main requirements (in functional form) is the MAIN_THEOREM given below. What we proved was a lemma, defined below as Msglemma, that constitutes a substantial part of MAIN_THEOREM. Msglemma covers MAIN_THEOREM only for the messages part of the behavior, but not for the final state part. Msglemma represents the most difficult part of MAIN_THEOREM.

```
MAIN_THEOREM := (StartSerp 's' 'Succ n' & ProperABS 's' & Liveness 's')
                => 'ActualBehavior s' = 'ExpBehavior s'

Msglemma := (ProperABS 's' & StartSerp 's' 'Succ n' & Liveness 's')
             => 'Msgs(ActualBehavior s)'='expected_messages s'
```

The macro step level was introduced to decompose the proof of the MsgLemma. MsgLemma was proved with the help of a set of intermediate lemmas—Vlemma, Clemma, and Slemma—one for each of the macro step functions. The relationship established by each of the intermediate lemmas is shown in Figure 10. An intermediate lemma associated with a macro step function states the following: The behavior of a trace of Steps that begins in a state in which the macro step function may be applied is the same as a trace in which an appropriate initial segment of Steps is replaced by the corresponding macro step. The formalizations of the intermediate lemmas are given below.

```
Vlemma := 'Behavior(trui(Step s))' =
            'Behavior (trui (VoteStep s))' , StartSerp 's' 'Succ n'
Clemma := 'Behavior (trui s)' =
            'Behavior (trui (CheckStep s))',
                              NormalLkup '(ABS_lkupof s)'
Slemma := 'Behavior (trui s)'=
          'Mcons (ABS_msgof s)(Behavior(trui(SBStep s)))',
                'is_send s'='True' & Liveness 's'
```

6.5 Definition of Expected Behavior

The expected behavior consists of two things: the list of messages that are approved during the SERP cycle and the state at the end of the SERP cycle. We will describe the definition of the `expected_messages` part of the behavior.

```
ExpBehavior s = <<expected_messages s, expected_final s>>
```

The expected messages are defined in two stages. First, the result of computing the voted SERP is defined by the function `(VotedSerp s)`. Then as a function of the voted SERP and the table of valid VIDs we can define the expected messages. When there is a broadcast pending, then there will be at most one message, and this is defined by the function `bcst_approved`.

```
expected_messages s {bpendingof (ABS_bpndingof s)} = bcst_approved s
expected_messages s = exp_msgs_from_check (VotedSerp s)(ABS_lkupof s)
```

The function that defines the messages in terms of the voted SERP and the lookup table is called `exp_msgs_from_check` since it is the list of messages that will be approved if the scoreboard reaches the checking state. To define it, we first form the expression `(all_clear vs fn num)`, which is the conjunction of the IBNF bits of the VSERP entries for the valid VIDs. This is used to determine whether a broadcast message is approved, and becomes the `bok` parameter to the function `checked_msgs`.

```
exp_msgs_from_check vs <<fn,num,i>> =
    checked_msgs (all_clear vs fn num) vs <<fn,num,Zero>>
```

The function `checked_msgs` defines the list of approved messages as follows. The expression `(message_list vserp lkup)` represents the list obtained by forming (as a function of the lookup table) the list `[v1,...,vn]` of valid VIDs, then applying the voted SERP to each `vi` to get its corresponding message, and then filtering out those messages that fail the OBNE or IBNF conditions. The function `sieve_dest`, which is applied next, deletes any message that has the same destination as a message earlier in the list. Finally, the function `bcst_truncate` deletes all messages after the first broadcast message and also deletes the first broadcast message if the `bok` is false.

```
checked_msgs bok vserp lkup =
   bcst_truncate bok (sieve_dest (message_list vserp lkup))
```

7 About the Verification

This section briefly describes the technique that was used to prove the correctness lemmas described in the previous section.

7.1 Msglemma

This lemma was by far the most difficult to prove of all the main lemmas. The proof relied on the previously proved `Vlemma`, `Clemma`, and `Slemma`, to allow us to view the trace of `Steps` as consisting of the macro steps defined by the functions `VoteStep`, `CheckStep`, and `SBStep`. In fact, since the state `s` is assumed to start in a `CHK` state, only `CheckStep` and `SBStep` will occur. Now the proof is by induction on the length of `(exp_msgs_nobp s)`, and proceeds by considering the possible sequences of macro steps. There are four possibilities:

1. After performing a `CheckStep`, the resulting state is `Idle`.
2. The `CheckStep` leads to a `Brdcsting` state, after which an `SBStep` leads to `Idle`.
3. The `CheckStep` leads to a `Sending` state, after which an `SBStep` leads to `Idle`.
4. The `CheckStep` leads to a `Sending` state, after which `SBStep` leads back to a `CHK` state.

In the first three cases, we were able to show directly, by expanding the definitions of `CheckStep`, `SBStep`, and `exp_msgs_nobp`, that the observed messages agree with the `exp_msgs_nobp`. In the fourth case, we had to appeal to our inductive hypothesis. To do this, we must show that the length of the list, `exp_msgs_nobp(SBStep(CheckStep s))`, is less than the length of `(exp_msgs_nobp s)`. In fact we must show that the list `exp_msgs_nobp(SBStep(CheckStep s))` is the tail of `(exp_msgs_nobp s)`, which was a major lemma in itself, the "check to check lemma," but could be proved by expanding the definitions of the functions involved and then invoking several general lemmas about the list operations (`map`, `filter`, `fromto`, `all`, etc).

7.2 Vlemma

To prove this lemma, it was sufficient to prove that applying the macro step, `VoteStep`, is the same as iterating `Step` some number, `n`, times, and that for no number `i < n`, does the i^{th} iterate of `Step` result in an `Idle` or `Sending` state. We proved this by making a stronger induction hypothesis, namely that the i^{th} iterate of `Step` is given by the expression `VotingStep i`. The function `VotingStep` has the property that `VotingStep n = VoteStep`, when `n` is the number of VIDs in the state, and for any `i < n` the result of (`VotingStep i`) is neither `Idle` nor `Sending`. Then, we proved by induction on `i` that `'iterate i Step'='VotingStep i'`.

7.3 Clemma

As for the `Vlemma`, it was sufficient to prove that the macro step, `CheckStep`, is the same as an n-fold iterate of `Step` for some number `n` and that for no number `i < n`, does the i^{th} iterate of `Step` result in an `Idle` or `Sending` state. Again we strengthened the induction hypothesis, by defining functions `CheckingStep i` and `checksteps` such that the i^{th} iterate of `Step` is given by the expression `CheckingStep i` and (for any state s) `CheckingStep (checksteps s) s = CheckStep s`, and for any `i < (checksteps s)` the result of `CheckingStep i` is neither `Idle` nor `Sending`. Then we proved by induction on `i` that `'iterate i Step'='CheckingStep i'`.

In this case, in order to derive the final `Clemma`, it was also necessary to prove that the function `checksteps` was well defined, which was proved by induction on the difference between the number of VIDs and the current index `i`.

7.4 Slemma

The proof of this lemma was similar to the proof of the `Clemma`, namely the functions (`SendingStep i`), (`BcstingStep i`), `sendingsteps`, and `bcstingsteps`, were defined so that it was possible to prove by induction on `i` that

1. `'iterate i Step s'='SendingStep i s' for i <= sendingsteps s`, and
2. `'iterate i Step s'='BcstingStep i s' for i <= bcstingsteps s`.

The functions (sendingsteps s) and (bcstingsteps s) are defined in terms of the least time at which a continue signal arrives from the network element. Thus they are only well defined assuming the rest of the network element behaves as it is supposed to. Therefore, Slemma is proved only under the Liveness assumption.

7.5 VC_ok_lemma

The VC_ok_lemma states that a certain equation must hold for each execution path, i.e., from cutpoint to cutpoint in the controller state diagram. We proved this lemma by considering each execution path. Spectool makes this proof easy by enumerating all the paths. Most of our effort was spent in writing the definition of the Step function. In this effort, Spectool and Clio were used to generate the definition of Step in a semi-automatic fashion.

8 Concluding Remarks

In conclusion, we summarize the main accomplishments of the Phase 1 effort.

1. We developed the functional specification for the Scoreboard in two forms. In the first form, the specification was expressed as a set of constraints on the behavior of the Scoreboard. In the second form, the expected behavior was specified as a single function of the initial state and the inputs to the Scoreboard.

2. We developed a high-level design for the Scoreboard using Spectool. The decomposition employed to construct the design was based on the algorithmic description and the VHDL design of the Scoreboard given in Morton's dissertation [9].

3. We proved a main theorem that established that the actual behavior of the design was equivalent to the expected behavior of the Scoreboard defined as a single function.

Formal verification is one of the most thorough forms of analysis that one can perform on a design. A benevolent side-effect of such an exercise is that one gains a deeper understanding of the problem and the design than is possible in other kinds of design analysis.

Constructing a precise abstract specification of the requirements is valuable as it serves as a contract between CSDL and ORA. It serves as a basis for conducting an effective

dialogue between CSDL and ORA to ensure that what we are verifying about the design is indeed what CSDL wants. Some of the questions that we have posed about the expected functionality of the Scoreboard in Section 4.4, particularly the ones related to broadcast message processing and Scoreboard initialization, came to light during the course of verification. We discovered several inconsistencies in our design and the specification of the expected behavior of Scoreboard. A significant number of the design errors were due to improper scheduling of the actions on the components that violated the delay requirements. Most of these errors were discovered fairly early during the verification process.

Although the verified design is not the actual CSDL design, the Phase 1 verification exercise was valuable in several respects. We expect the experience gained to be useful in performing the second phase of the task. A significant part of the current proof can be reused in proving the actual design because CSDL's detailed Scoreboard design is expected to be similar to the one we verified. Since our proof was layered into several levels, the structure of the proof and several of the lemmas that were needed in the proof at the higher end were independent of the design details and, hence, can be reused in the verification of a new design.

A subtle error that was discovered only late in the verification process was a possible inconsistency in the manner in which messages to invalid destinations were processed by our design. The design was enforcing the Scoreboard requirement that no two messages may be approved to the same destination even for invalid destination VID's. The outcome was that if there were two or more distinct messages that were destined to the same invalid destination, then only one of them, the one with the smallest source VID, would get sent (of course, after modifying the destination to "null"). We had mistakenly assumed that a certain way of initializing an internal table of the Scoreboard would take care of the checking for invalid destinations as required. We had ignored the possibility of the existence of two messages destined to identical invalid destinations.

Our planned course of action for Phase 2 is as follows. First, we must have the CSDL team read our functional requirements specification to ensure that the specification is accurate and also adequate. The second step is to study and understand the actual Scoreboard design. To perform this step effectively, we plan to have a member of ORA's Scoreboard verification team spend a suitable period of time at CSDL to interact with the CSDL engineer who is in charge of developing the detailed design. The CSDL team has decided to build the Scoreboard on a PC board as a set of interconnected IC blocks, where

every block is an off-the-shelf component (such as a memory device) or is built using standard cells (such as gate arrays and PLA's). We plan to verify the Scoreboard at the IC block level. For this, we have to develop a formal specification of the logic implemented by each of the blocks in the design. Then, we must formally verify that the composition of the blocks meets the functional requirements specification.

References

[1] William R. Bevier and William D. Young. "Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit". NASA Contractor Report 182099, NASA Langley Research Center, Hampton, VA 23665-5225, November 1990. Authors' affiliation: Computational Logic, Inc., Austin, Texas.

[2] M. Bickford, C. Mills, and E.A. Schneider. "Clio: An Applicative Language-Based Verification System". Technical Report TR 89-13, ORA Corporation, 301A Harris B. Dates Drive, Ithaca, NY 14850, September 1989.

[3] Ricky W. Butler. "NASA Langley's Research Program in Formal Methods". In *Proceedings of the Sixth Annual Conference on Computer Assurance*, pages 157–162, June 1991.

[4] Avra Cohn. "Correctness Properties of the Viper Block Model: The Second Level". Technical Report 134, Computer Laboratory, University of Cambridge, Cambridge, U.K., May 1988.

[5] R.E. Harper et al. "The Army Fault Tolerant Architecture Conceptual Study". NASA Contract NAS1-18565 Task 14 Report, The Charles Stark Draper Laboratory, 555 Technology Square, Cambridge, MA 02139, August 1991.

[6] R.E. Harper, J. Lala, and J. Deyst. "Fault Tolerant Parallel Processor Architecture Overview". In *"18th International Symposium on Fault Tolerant Computing"*, pages 252–257, June 1988.

[7] Warren A. Hunt. "FM8501: A Verified Microprocessor". In *IFIP WG 10.2 Workshop, From HDL to Guaranteed Correct Circuit Designs*, pages 85–114. North-Holland Publishing Co., 1986.

[8] Roger Lipsett, Carl Schaefer, and Cary Ussery. *"VHDL: Hardware Description and Design"*. Kluwer Academic Publishers, 1989.

[9] Dennis Morton. "Hardware Modeling and Top-down Design Using VHDL". Technical Report CSDL-T-1082, The Charles Stark Draper Laboratory, 555 Technology Square, Cambridge, MA 02139, June 1991. MS Thesis, MIT, Cambridge, MA 02139.

[10] M. Pease, R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". *JACM*, 27(2):228–234, April 1980.

[11] Natarajan Shankar. "Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm". NASA Contractor Report 4386, NASA, Langley Research Center, Hampton, Virginia 23665-5225, July 1991. Authors' affiliation: SRI International, Computer Science Laboratory, Menlo Park, CA.

[12] Mandayam Srivas. "Bridging the Formal Methods Gap: A Computer-Aided Verification Tool for Hardware Designs". In *COMPCON91*, San Francisco, CA, February 25-28 1991.

[13] Mandayam Srivas and Mark Bickford. "Formal Verification of a Pipelined Microprocessor". *IEEE Software*, September 1990.

[14] Mandayam Srivas and Mark Bickford. "Spectool: A Computer-Aided Verification Tool for Hardware Designs, Final Report, Contract No. F30602-89-D-0096". Rome Laboratory, Griffis AFB, NY 13441, 1991. Authors' affiliation: ORA Corporation, 301A Dates Drive, Ithaca, NY 14850.

[15] Mandayam Srivas and Mark Bickford. "Verification of the FtCayuga Fault-Tolerant Microprocessor System Volume 1: A Case Study in Theorem Prover-Based Verification". NASA Contractor Report 4381, NASA Langley Research Center, Hampton, VA 23665-5225, 1991. Authors' affiliation: ORA Corporation, 301A Dates Drive, Ithaca, NY 14850.

A The Scoreboard Controller Schedule

```
BGN  0 = [(opin = reset) |
          (opin = clear_timeouts) -> reset_tm TMO]
BGN  3 = [reset_bok BOK, reset_index LKUP, advance TIMER]

GET  0 = [read_ct CT]
GET  2 = [readserp SERP, get TMO]
GET  3 = [advance TIMER]

VOT  0 = [vote VOTE]
VOT  1 = [choose1_of2 MX2, set_vs VS, and_bok BOK, set TMO]
VOT  2 = [choose1 MX, store VSERP]
VOT  3 = [advance TIMER, next_index LKUP]

CHK  0 = [choose1 MX, read VSERP]
CHK  1 = [gen_msg MSG, bt_set OBNE]
CHK  2 = [check_validity VALID, choose3 MX, read VSERP]
CHK  3 = [(valid & bcst & c_obne) -> set_bp BDCST,
          (~valid) -> make_null_dest MSG,
          choose2_of2 MX2, set_vs VS, advance TIMER]
CHK  4 = [next_index LKUP]

SND  0 = [clear_ibnf VS]
SND  1 = [choose3 MX, store VSERP]
SND  3 = [advance TIMER]

BROD  0 = [clear_bp BDCST]
BROD  1 = [choose2 MX, read VSERP]
BROD  2 = [gen_msg MSG]
BROD  3 = [advance TIMER]

SNDB  3 = [advance TIMER]
```

B The Scoreboard Components Actions

(The delays of the actions are indicated in parantheses.)

```
BOK reset_bok (1), and_bok (1)

BDCST clear_bp (1), set_bp (1)

CT read_ct (1)

LKUP reset_index (1), next_index (1), set_table (1), reset_table (1)

MX choose1 (0), choose2 (0), choose3 (0)

MX2 choose1_of2 (0), choose2_of2 (0)

MSG gen_msg (1), make_null_dest (1)

OBNE bt_set (1)

SERP readserp (1), writeserp (1)

TIMER reset_timer (1), advance (1)

TMO reset_tm (1), get (1), set (1)

VALID check_validity (1)

VOTE vote (1)

VS clear_ibnf (1), set_vs (1)

VSERP store (1), read (1)
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May, 1992	Contractor Report

4. TITLE AND SUBTITLE
Moving Formal Methods into Practice: Verifying the FTPP Scoreboard: Phase 1 Results

5. FUNDING NUMBERS
C NAS1-18972
WU 505-64-10-05

6. AUTHOR(S)
Mandayam Srivas and Mark Bickford

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ORA Corporation
301 Harris Dates Drive
Ithaca, NY 14850-1313

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-189607

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Paul S. Miner
Task 5 Report

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This report documents the Phase 1 results of an effort aimed at formally verifying a key hardware component, called Scoreboard, of a Fault-Tolerant Parallel Processor (FTPP) being built at Charles Stark Draper Laboratory, Inc. (CSDL). The Scoreboard is part of the FTPP virtual bus that guarantees reliable communication between processors in the presence of Byzantine faults in the system. The Scoreboard implements a piece of control logic that approves and validates a message before it can be transmitted. The goal of Phase 1 was to lay the foundation for the Scoreboard verification. A formal specification of the functional requirements and a high-level hardware design for the Scoreboard were developed. The hardware design was based on a preliminary Scoreboard design developed at CSDL. A main correctness theorem, from which the functional requirements can be established as corollaries, has been proved for the Scoreboard design. The goal of Phase 2 is to verify the final detailed design of Scoreboard. This task is being conducted as part of a NASA-sponsored effort to explore integration of formal methods in the development cycle of current fault-tolerant architectures being built in the aerospace industry.

14. SUBJECT TERMS Formal requirements specification, Fault-tolerant parallel computer, Byzantine resilience, Computer-aided hardware verification, Theorem prover-based verification

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	